Cairo University

AUTOMATING THE GENERATION AND TYPESETTING OF ARABIC SCRIPT

by

Sherif Samir Hassan Mansour

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2015

AUTOMATING THE GENERATION AND TYPESETTING OF ARABIC SCRIPT

by

Sherif Samir Hassan Mansour

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Under the Supervision of

Dr. Hossam A. H. Fahmy
Associate Professor
Electronics and Communications Engineering Department
Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2015

AUTOMATING THE GENERATION AND TYPESETTING OF ARABIC SCRIPT

by

Sherif Samir Hassan Mansour

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Approved by the
Examination Committee

Associate Prof. Hossam A. H. Fahmy, Thesis Advisor

Prof. Mohsen AbdelRazik Rashwan, Internal Member

Prof. El-Sayed Mostafa Saad, External Member
(Professor at the Faculty of Engineering, Helwan University)

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2015

Engineer: Sherif Samir Hassan Mansour
Date of Birth: 09/02/1986
Nationality: Egyptian
E-mail: sherif.s.mansour@gmail.com
Phone: +201227120033
Address: Building 16, Group 87, Al-Rehab City, Cairo

Registration Date: 1/10/2010
Awarding Date: / /
Degree: Master of Science
Department: Electronics and Communications Engineering

Supervisors: Associate Prof. Hossam A. H. Fahmy

Examiners: Associate Prof. Hossam A. H. Fahmy (Thesis advisor)
Prof. Mohsen AbdelRazik Rashwan (Internal examiner)
Prof. El-Sayed Mostafa Saad (External examiner)

Title of Thesis:

Automating the Generation and Typesetting of Arabic Script

Key Words: Arabic, Fonts, TeX, AlQalam, Context Analysis.

Summary:

This research continues to build on the previous work in “AlQalam” project for font testing and development.
In order to get a usable font package, we started by building a font package using Metafont and resolved the major bugs that we faced while using it to write Arabic documents.
Then, we developed a generic context analysis algorithm for rich fonts using C language and ported it to Lua language to be used with “AlQalam” font.
The ported code has been embedded with the font using the new features offered by LuaTeX to enable users to generate texts using “AlQalam” font quickly and easily.

Acknowledgments

I would like to thank Allah the Most Gracious, the Most Merciful for all his blessings and bounties.

I would like to dedicate this thesis to the soul of my father, my first teacher; I know that he would be very happy and proud to see me following his steps.

To my mother and my sister for their continuous support and encouragement.

I would like to thank Dr. Hossam Fahmy for being my supervisor, mentor and brother. He didn't only help me technically and encouraged me through the research and Master thesis preparation journey but also he helped me a lot on the personal aspect of life.

To Ahmed Ismail, Amin Maher and Khaled Nouh, my friends, colleagues and companions in the Masters journey for their support, knowledge sharing and car sharing in our daily trip during the Pre-Masters year.

To Osama Alaa and Hend Maher for their help during the font debugging phase and their help in fixing the font bugs and getting an enhanced font package.

To Robin Laakso and Karl Berry for their help during my trip to Boston and their help publishing my first paper in TUGBoat.

To Amy Hendrickson and Jonathan cook for their generous hospitality during my stay in Boston for the TUG 2012 conference.

To Amr Abdulzahir and Islam Beltagy for their help planning the trip to Boston and taking the responsibility of tour guiding me in Boston.

To Khaled Hosny for his help during the LuaTEX exploration phase.

To the people who have influenced my life in many ways.

Contents

List of Figures

List of Abbreviations and Glossary of Terms

ASCII	American Standard Code for Information Interchange.
DPI	Dots Per Inch.
DVI	DeVice Independent.
GF	An extension used by Metafont to represent a generic font file.
LuaTEX	An extended version of pdfTEX using Lua as an embedded scripting language.
Metafont	A description language used to define vector fonts and also the name of its interpreter.
MetaPost	A programming language and the interpreter of the MetaPost programming language.
OTF	OpenType Font.
PDF	Portable Document Format.
pdfTEX	An extension of the typesetting program TEX .
PK	An extension used by Metafont to represent a packed font file.
QC	Quality Control.
TEX	A typesetting system designed and mostly written by Donald Knuth.
TFM	TEX font metric is a font file format used by the TEX .
TTF	TrueType Font.
TUG	TEX Users Group.
TUGboat	TEX Users Group journal.
Unicode	A computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.
UTF-8	(U from Universal Character Set + Transformation Format—8-bit) is a character encoding capable of encoding all possible characters (called code points) in Unicode.
WYSIWYG	What You See Is What You Get.

Abstract

This research continues to build on the previous work in AlQalam project for font testing and development. In order to get a usable font package, we started by building a font package using Metafont. This included debugging and fixing issues with Metafont files of the developed letters.

Then, we moved the package to TEX in order to start using it. As an initial phase, we were calling the letter forms manually and this helped us to discover more issues with the font that needs fixing and enhancing like problems joining the letters correctly and problems of missing letter forms. A new phase of font debugging started to fix these issues.

After correcting and fixing the problems, we developed a generic third-party program using C language to handle the context analysis part of any rich Arabic font including AlQalam.

We tested the algorithm with AlQalam font and wanted to embed it somehow with AlQalam to facilitate using the font without the hassle of running a third party program and making lots of steps to get a document written with AlQalam.

Accordingly, we investigated the new features introduced in LuaTEX and found that it provides the required capabilities to load Lua functions and use them to override the default operations of the TEX engine through callbacks.

Hence, we ported our generic context analysis C code to Lua language and loaded it inside LuaTEX and overrode the normal input buffer TEX operations to manipulate the buffer contents and do our custom context analysis work and get sentences written in AlQalam font with the correct letter forms according to the context using a normal LuaTEX editor to input Arabic text.

Chapter 1

Introduction

The Arabic alphabet is used to write many languages in many places around the world. Also, Arabic is of great importance to Muslims (about a quarter of the world's population), as it is the liturgical language of Islam. The most distinctive features of the Arabic alphabet are that it includes 28 letters and is written horizontally from right to left in cursive style, i.e., many or all letters in a word are connected. Figure 1.1 shows the Arabic alphabets in their morphological order

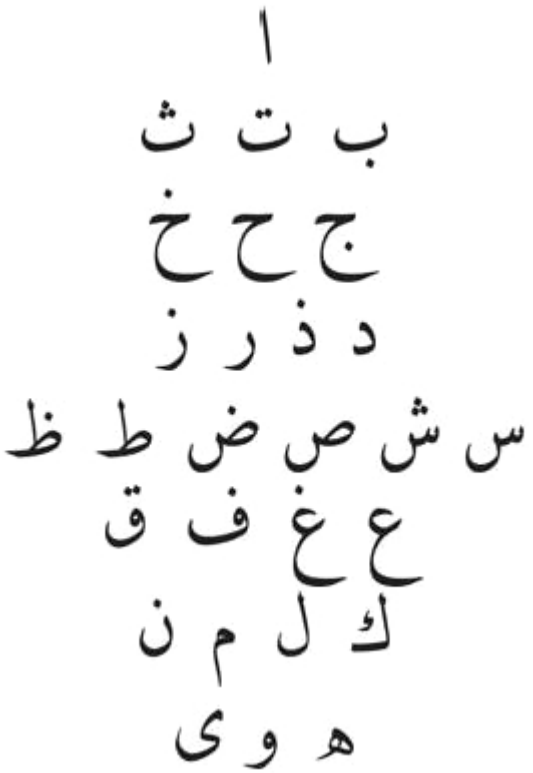


Figure 1.1: Arabic alphabets

Arabic spelling is not fully alphabetic: only short consonants and long vowels are written with the primary character set. For elaborate spelling or casual disambiguation, a set

of secondary characters exists. They are written above or below a primary character, e.g. The hamza diacritic and its various supporting letters:

ء أ إ ئ ي ؤ

FATHA to mark the vowel /a/:

DHAMMA to mark the vowel /u/:

KASRA to mark the vowel /i/:

FATHATAN to mark the vowel /a/ +n:

DHAMMATAN to mark the vowel /u/ +n:

KASRATAN to mark the vowel /i/+n:

Arabic is used to write: Arabic, Arabic (Algerian), Arabic (Egyptian), Arabic (Lebanese), Arabic (Modern Standard), Arabic (Moroccan), Arabic (Syrian), Äynu, Azeri, Baluchi, Beja, Bosnian, Brahui, Crimean Tatar, Dari, Gilaki, Hausa, Kabyle, Karakalpak, Konkani, Kashmiri, Kazakh, Khowar, Kurdish, Kyrgyz, Malay, Marwari, Mandekan, Mazandarani, Morisco, Pashto, Persian/Farsi, Punjabi, Rajasthani, Salar, Saraiki, Shabaki, Sindhi, Somali, Tatar, Tausūg, Turkish, Urdu, Uyghur, Uzbek and a number of other languages.

Arabic is a Semitic language with more than 422 million speakers in Afghanistan, Algeria, Bahrain, Chad, Cyprus, Djibouti, Egypt, Eritrea, Iran, Iraq, Jordan, Kenya, Kuwait, Lebanon, Libya, Mali, Mauritania, Morocco, Niger, Oman, Palestinian West Bank and Gaza, Qatar, Saudi Arabia, Somalia, Sudan, Syria, Tajikistan, Tanzania, Tunisia, Turkey, UAE, Uzbekistan and Yemen and used by more than 1.5 billion Muslims around the world according to UNESCO 2014 statistics.

The long vowels /a:/, /i:/ and /u:/ are represented by the letters 'alif, yā' and wāw respectively. Vowel diacritics, which are used to mark short vowels and other special symbols appear only in the Qur'an. They are also used, though with less consistency, in other religious texts, in classical poetry, in books for children and foreign learners, and occasionally in complex texts to avoid ambiguity. Sometimes the diacritics are used for decorative purposes in book titles, letterheads, nameplates, etc.

Arabic has six major writing styles: Kufi, Thuluth, Naskh, Riq'aa, Deewani, and Ta'liq. Naskh style is the most commonly used for printing, both of traditional texts such as the Muslim Holy Book (the Qur'an) as well as contemporary publications. Figures 1.2 and 1.3 show the major Arabic writing styles

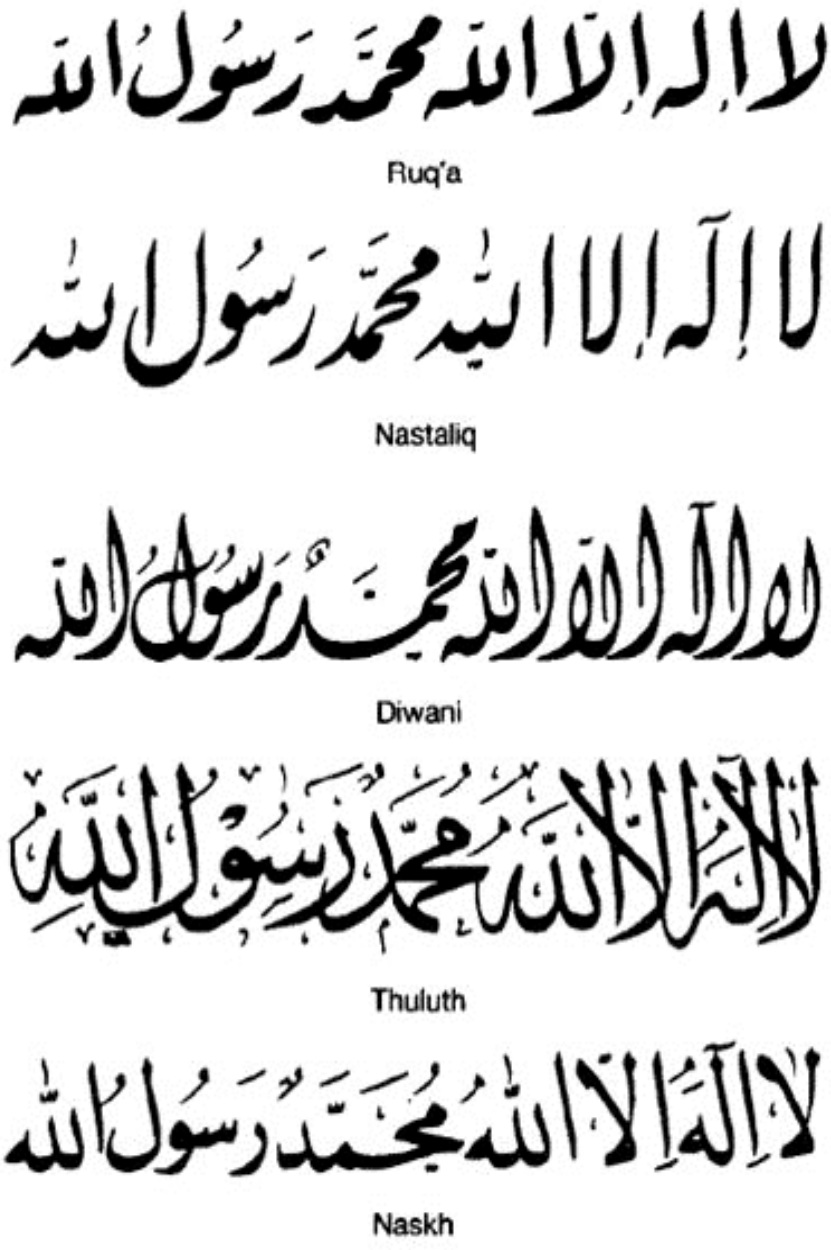


Figure 1.2: Major Arabic writing styles

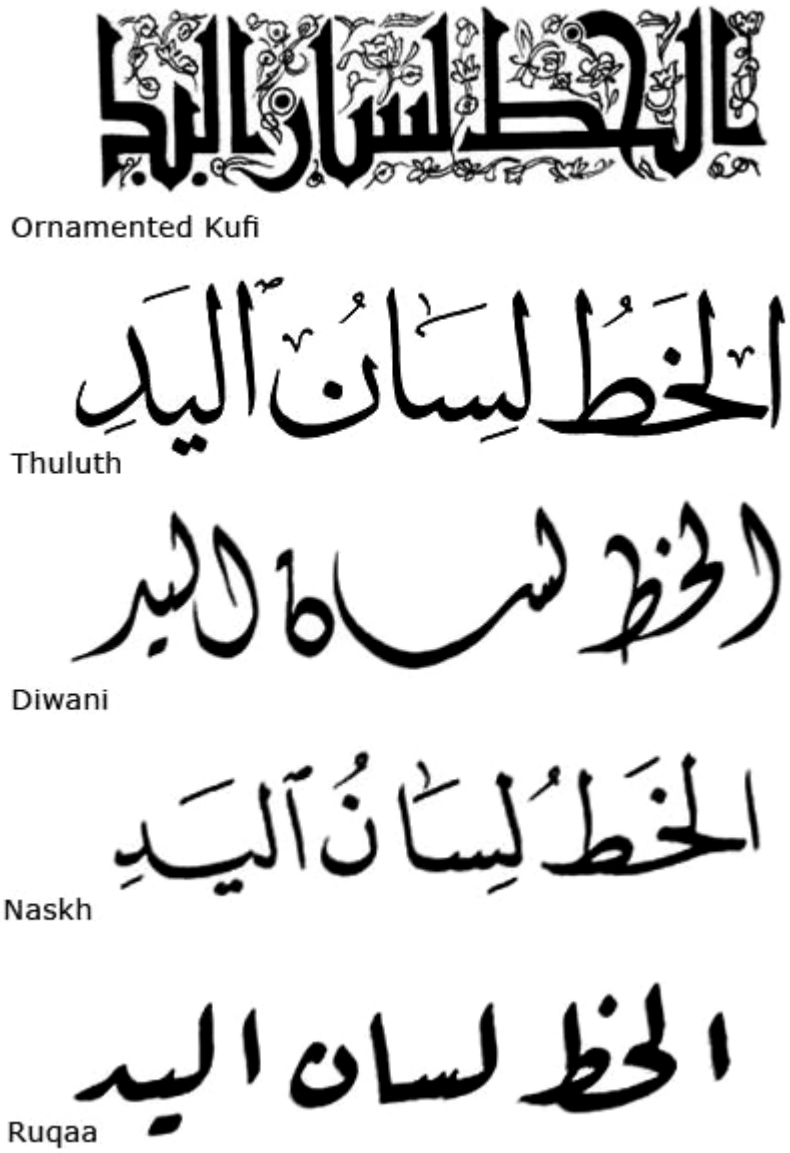


Figure 1.3: Major Arabic writing styles

Some trials were made to generate documents using Nastaleeq style which is the default writing style for Urdu [1]. In our study we'll focus on generating documents using Naskh style which is more commonly used as mentioned before.

1.1 Challenges in producing high-quality Arabic fonts

The main challenge of producing high-quality Arabic fonts is that Arabic calligraphy is an art. The rules to follow when composing Arabic texts have great flexibility in choosing different variations of letter forms and constructing complex ligatures. These variations and ligatures add an aesthetic touch to the script and also justify the text as needed. Every Arabic letter may have four basic forms depending on its location in the word: initial, medial, final, and isolated.

Figure 1.4 shows the different forms of the "Baa" letter as an example.

Figure 1.4: From right to left: initial, medial, final, and isolated forms of "Baa"

Every letter form may have different variations to be used depending on the preceding or succeeding letter. Figure 1.5 (taken from [2]) for example shows the different variants of the forms of "Baa". The first shape on the right is the isolated 'Baa" form variant. The second, third and sixth shapes are initial form variants of "Baa" when rising letters like "Alef" precede it, letters like "Jeem" precedes it and letters like "Seen" precede it, respectively. The fourth and fifth shapes are medial form variants of "Baa" when letters like "Haa" and letters like "Raa" precede it respectively. The seventh shape is the general medial form variant of "Baa" and the last shape on the left is a final form variant of "Baa".

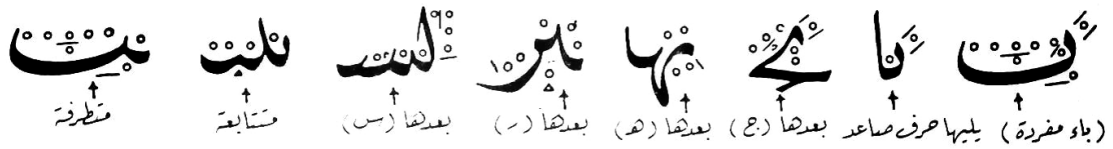


Figure 1.5: Letter "Baa" form variations

Figure 1.6 shows a part of Mushaf Al-Madinah (The Holy Qur'an - Madinah print) [3] as a rich example of the usage of different combinations of variations with different elongations. This printing, like most printings of the Qur'an, was hand-written by a calligrapher and is not typeset. Computer typesetting (and to a large extent also traditional mechanical typesetting) are well behind what a calligrapher can produce for such complex texts. The underlined shapes represent the same Arabic letter, which is "Kaf". Notice the different forms used depending on the location of the letter in the word and notice that some forms have different variations depending on the elongation requirements for line breaking, text justification and preceding or succeeding letters.

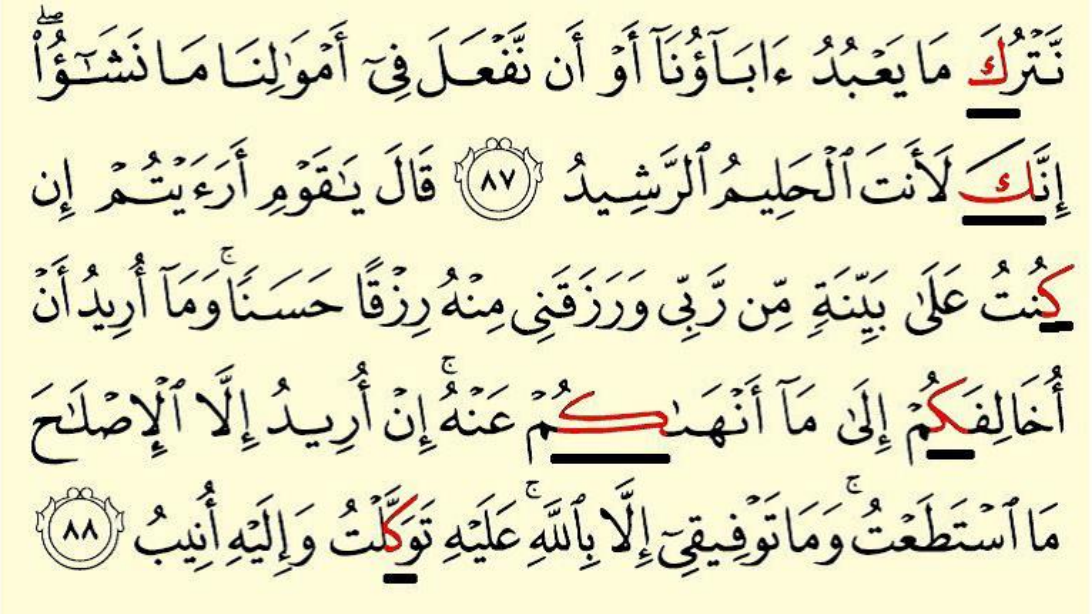

نَتْرُكَ مَا يَعْبُدُ ءَابَآؤُنَآ أَوْ أَن نَّفْعَلَ فِىٓ أَمْوَٰلِنَا مَا نَشَٰٓؤُا۟ ۖ
إِنَّكَ لَأَنتَ ٱلْحَلِيمُ ٱلرَّشِيدُ ﴿٨٧﴾ قَالَ يَٰقَوْمِ أَرَءَيْتُمْ إِن
كُنتُ عَلَىٰ بَيِّنَةٍ مِّن رَّبِّى وَرَزَقَنِى مِنْهُ رِزْقًا حَسَنًا ۚ وَمَآ أُرِيدُ أَنْ
أُخَالِفَكُمْ إِلَىٰ مَآ أَنْهَىٰكُمْ عَنْهُ ۚ إِنْ أُرِيدُ إِلَّا ٱلْإِصْلَٰحَ
مَا ٱسْتَطَعْتُ ۚ وَمَا تَوْفِيقِىٓ إِلَّا بِٱللَّهِ ۚ عَلَيْهِ تَوَكَّلْتُ وَإِلَيْهِ أُنِيبُ ﴿٨٨﴾

Figure 1.6: An example from surat Hud: forms of "Kaf"

1.2 The AlQalam font and its features

Several trials were performed to produce high-quality Arabic fonts. One of them was the AlQalam ("The Pen" in Arabic) project that was started in 2006 under the supervisor's supervision [4]. AlQalam's target was to simulate an Arab calligrapher's pen for Naskh style (as used to typeset the Qur'an printing, for example). AlQalam then might be used in typesetting any traditional texts as well as any generic publications (including scientific ones) in the languages using the Arabic script.

At first, AlQalam grew out of modifications to ArabTEX [5]. Modifications were mainly intended to respond to the specific needs of typesetting the Qur'an such as adding pause signs, some additional diacritics (marks used as phonetic guides) and the abilities to stack them on top of each other, scale them and correctly position them on the word. Also, some modifications to the pen used were made to improve the shape of some letters and symbols.

In 2008, a new font project for AlQalam was started [6]. That font is meta-designed such that each character is described by a number of parameters to allow the creation of many variants that connect with the surrounding characters correctly. Those variants may be different in shape and in their amount of elongation. Starting from this period many modifications were made and new features added.

Here is a brief points for AlQalam's font features up till now:

- All font shapes are meta-designed using Metafont to enable greater flexibility while joining glyphs together and provide smoother letter extensions.
- It contains the generic four different forms of Arabic letters (initial, medial, final, and isolated).
- It also contains different parameterized shapes for letter forms (the main source of the form variants is Mushaf Al-Madina).

- It is based on the concept of primitives (reusable glyphs). For example, Figure 1.7 shows the Waw head primitive (the small circle). This Waw head is reused in combination with the body (skeleton) of letter "Baa" to produce the letter "Faa". Also, the "Waw" head can be reused in combination with the body of the letter "Noon" to produce the letter "Qaf".
- The font supports vertical placement of glyphs: Various ligatures have been added to the font to support complex vertical placement combinations as shown in Figure 1.8.
- Kerning: Borders of letter boxes have been adjusted to support kerning as shown in Figure 1.9.
- Joining glyphs with kashidas: the kashida is the most widely used glyph to join letters. AlQalam implements the kashida as a dynamic smooth glyph, as shown in Figure 1.10. This is preferable to the current standard fonts that implement the kashida as a static fixed length glyph, as shown in Figure 1.11.
- Parameterized diacritics: A complete set of parameterized diacritics that can be elongated according to the width of the associated letter is available, as shown in Figure 1.12.
- Mathematical symbols: AlQalam is one of three fonts that have a complete set of Arabic math symbols at the time of writing. (The other two fonts are RyDArab [7] and Mathfont [8].) Examples for Arabic equations generated using AlQalam are shown in Figure 1.13.

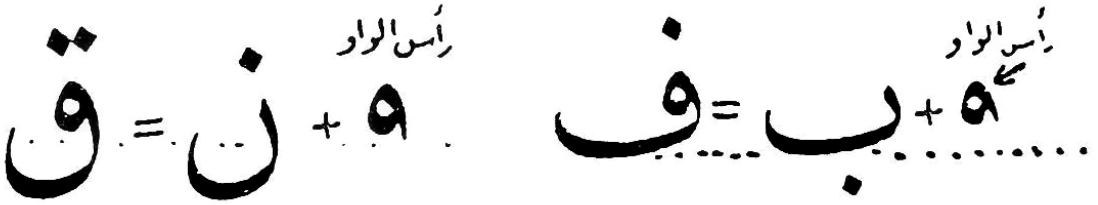


Figure 1.7: The "Waw" head primitive

Figure 1.8: Vertical placement of glyphs

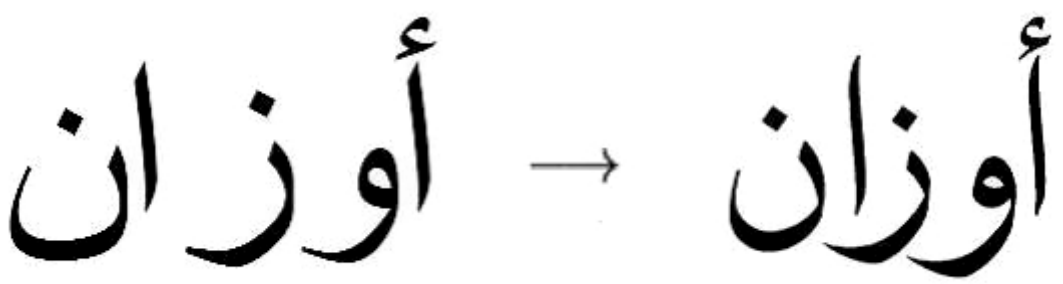


Figure 1.9: Kerning

Figure 1.10: Joining glyphs with smooth, dynamic kashidas

Figure 1.11: Static fixed length kashidas

Figure 1.12: Parameterized diacritics

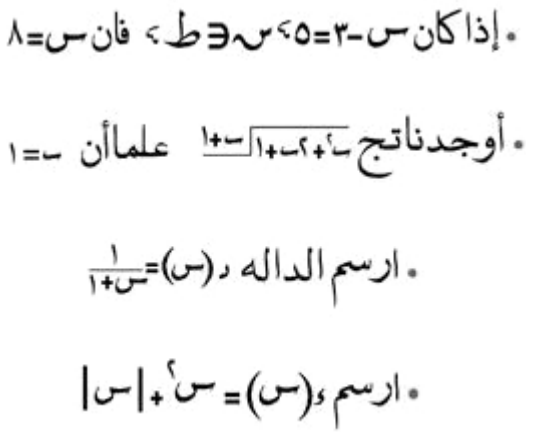


Figure 1.13: Set of Arabic mathematical equations typeset with AlQalam

1.3 Goal of Thesis

After getting through the information about Arabic language, writing and typesetting in addition to getting an idea about AlQalam project state and current features, we came to state our goal from this thesis. Having a very rich font features like those presented in AlQalam requires special support that will enable the user to make use of all the features.

A latin font can be integrated easily to an editor like Microsoft Word and TeX as the regular latin fonts use the same context analysis and line breaking algorithms which are, when compared to Arabic, may seem less difficult as most of the Latin fonts are not cursive and are written in a separate letters fashion. This makes it rarely needed to have a context analysis algorithm for Latin fonts. Also, the line breaking algorithm needed to justify Latin depends mainly on spaces stretching which makes it easier than in the complicated case of Arabic which depends on spaces, elongation of letters, ligatures and changing letter forms. [9]

So the goal of this thesis is to provide AlQalam font and any other Arabic font with needed automation algorithms to make it usable and fully functional on the user side. This work was done on several phases and there is still some work that needs to be done.

1.4 Motivation

Arabic is the language of the Holy Qur'an and main language of Islam. We saw the need of serving it to enrich the Arabic resources in the digital era as the state of the digital typesetting for Arabic is somewhat poor as the widely available Arabic fonts serves the very basic needs of the Arabic language and also doesn't fully represent its beauty and rich features.

1.5 Publications

Sherif Mansour and Hossam A. H. Fahmy, "Experiences with Arabic font development", TUGboat, vol. 33, no. 3, pp. 295--298, 2012.

Chapter 2

State of the Art

There are a lot of ways to typeset Arabic using digital typesetting methods. Here we'll discuss the most common methods and their features.

2.1 ArabTEX

ArabTEX which is developed by Klaus Lagally was one of the first trials to typeset Arabic inside TEX/LATEXenvironments. ArabTEX is a package extending the capabilities of TEX / LATEX to generate the Arabic writing. It can take romanized ASCII to produce quality ligatures for Arabic. ArabTEXcharacters are placed within a TEX /LATEX document using the command `\RL{ … }` or the environment `\begin{RLtext}` … `\end{RLtext}`. It consists of a TeX macro package and an Arabic font in several sizes, presently only available in the Naskhi style. ArabTEX will run with Plain TEX and also with LATEX ; other additions to TEX have not been tried. ArabTEX is primarily intended for generating the Arabic writing, but the scientific transcription can be also easily generated. For other languages using the Arabic script limited support is available.

Examples: `\RL{al-salAm `alaykum}` produces what is shown in Figure 2.1

السلام عليكم

Figure 2.1: AlSalam Alikom using ArabTEX

and using the following piece of code:

```
\documentclass[12pt]{article}
\usepackage{arabtex}
\begin{document}
\setarab
\fullvocalize
\transtrue
\arabtrue
\begin{RLtext}
```

```
bismi al-ll_ahi al-rra.hm_ani al-rra.hImi
\end{RLtext}
\end{document}
```

yields "Basmalah" which is shown in Figure 2.2

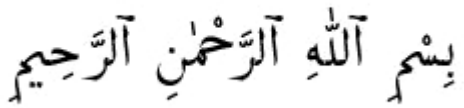


Figure 2.2: Basmalah using ArabTEX

We think that the use of the ASCII transliteration is quite difficult and it is still not very flexible and doesn't show the art and beauty of Arabic calligraphy. In case user don't want to deal with utf-8 encoding in his document, then ArabTeX package may be a good choice.

2.2 RyDArab

The package RyDArab extends TEX to handle mathematics in an Arabic presentation. That means expressions with specific symbols flowing from right to left, according to the Arabic writing, as they can be found in Arabic mathematical handbooks. The system consists of a set of TEX macro packages, some additional extensions and a family of symbol fonts. It will run under the Plain TEX or LATEX formats. Still depending on a modified version of ArabTEX , RyDArab adapts the concept of transliteration of Latin ASCII characters to generate Arabic words and math commands to generate Arabic mathematical expressions. [10]

Example:

```
\funwithdots=
$\sin c + \tan s$
```

produces what is shown in Figure 2.3

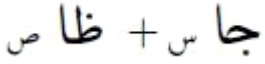


Figure 2.3: An equation using RyDArab

and

```
${\newfunc{SGr}}(c) = \cos(c{{}\sp 2}) - 6$
```

$$\text{صغر}\,(\text{س}) = \text{جتا}\,(\text{س}^2) - 6$$

Figure 2.4: Another equation using RyDArab

produces what is shown in Figure 2.4

As the reader notes that this package focuses only on mathematical symbols and expressions and also is using the transliteration concept which makes it some how difficult to input and use.

2.3 XeLaTeX /XeTeX

Using XeLaTeX or XeTeX engines to compile Arabic documents, rather than other engines, was probably the best option because it allows you to use any Arabic system fonts you have and input your text using utf-8 encoded Arabic letters. [11]

There are a number of packages that will work with XeLaTeX to get Arabic into tex documents. One of them is the Polyglossia package which also supports mixing Arabic and Latin inline.

Here is an example code for using XeLaTeX with Polyglossia shown in figure 2.5:

```
\documentclass{book}

\usepackage{setspace}

\usepackage{fontspec}
\usepackage{polyglossia}

\setmainlanguage{english}
\setotherlanguage{arabic}

\newfontfamily\arabicfont[Script=Arabic,Scale=1.1]{Traditional Arabic}

\begin{document}

Some latin text and inline arabic: \textarabic{السلام عليكم}

And for larger blocks of text you can use Arabic environment:

\begin{Arabic}

تحتوي العربية على 28 حرفًا مكتوبًا. ويرى بعض اللغويين أنه يجب إضافة حرف الهمزة إلى حروف
مثلها اللغة الفارسية - العربية، ليصبح عدد الحروف 29. تكتب العربية من اليمين إلى اليسار
ومن أعلى الصفحة إلى أسفلها - والعبرية وعلى عكس الكثير من اللغات العالمية.

\end{Arabic}

\end{document}
```

Figure 2.5: Arabic input using XeLaTeX and Polyglossia

and here is the output shown in figure 2.6:

Some latin text and inline arabic: السلام عليكم

And for larger blocks of text you can use Arabic environment:

تحتوي العربية على 28 حرفًا مكتوبًا. ويرى بعض اللغويين أنه يجب إضافة حرف الهمزة إلى حروف العربية، ليصبح عدد الحروف 29. تكتب العربية من اليمين إلى اليسار – مثلها اللغة الفارسية والعبرية وعلى عكس الكثير من اللغات العالمية – ومن أعلى الصفحة إلى أسفلها.

Figure 2.6: Arabic using XeLaTeX and Polyglossia

The previous method can be used with any Arabic font installed on the system.

There is an option for users who want to use the XeLaTeX engine and input non-utf8 text which is the ArabXeTeX package.

ArabXeTEX package provides a convenient ArabTEX -like user-interface for typesetting languages using the Arabic script in XeLaTeX, with flexible access to font features. Input in ArabTEX notation can be set in three different vocalization modes or in roman transliteration. Direct UTF-8 input is also supported. So the main job of this package is to parse and convert ArabTEX input to Unicode and then pass the output to XeLATEX engine.

2.4 Babel and Arabi

The standard distribution of LATEX contains a number of document classes that are meant to be used, but also serve as examples for other users to create their own document classes. These document classes have become very popular among LATEX users. But it should be kept in mind that they were designed for American tastes and typography. At one time they contained a number of hard-wired texts. Here comes the role of babel, a package that makes use of the new capabilities of TEX version 3 to provide an environment in which documents can be typeset in a non-American language or in more than one language. [12]

The Arabi package Arabic script support for TEX without the need for an external pre-processor and adds one of the major multilingual typesetting capabilities to Babel by adding support for the Arabic and Farsi languages.

Arabi comes with many good quality free fonts, Arabic and Farsi, and may also use commercial fonts. It supports many 8-bit input encodings (namely, CP-1256, ISO-8859-6 and Unicode UTF-8) and can typeset classical Arabic poetry. [13]

The Arabi package provides an Arabic and Farsi script support for TeX without the need of any external pre-processor, and in a way that is compatible with babel. The bi-directional capability supposes that the user has a TEX engine that knows the four primitives `\beginR`, `\endR`, `\beginL` and `\endL`. That is the case in both the TEX --XeT and e-TEX engines. Arabi will accept input in several 8-bit en-cod-ings, including UTF-8. Arabi can make use of a wide variety of Arabic and Farsi fonts, and provides one of its own. PDF files generated using Arabi may be searched, and text may be copied from them and pasted elsewhere.

Here is an example for Arabi input file in figure 2.7
and its output at figure 2.8.

```
\documantclass{article}
\usepackage[ cp1256 ]{inputenc}
\usepackage[ arabic ,english]{babel}
\begin{document}
 \selectlanguage{arabic}
بسم الله الرحمن الرحيم .
\\
الفصل السادس عشر في الاستخارة

في صحيح البخاري عن جابر قال كان رسول الله يعلمنا الاستخارة في الامر كما يعلمنا السورة من القران اذا هم احدكم بالامر فليركع ركعتين من غير الفريضة ثم ليقل اللهم اني استخيرك بعلمك واستقدرك بقدرتك واسالك من فضلك العظيم فانك تقدر ولا اقدر وتعلم ولا اعلم وانت علام الغيوب اللهم ان كنت تعلم ان هذا الامر ويسمى حاجته خير لي في ديني ومعاشي وعاقبة امري فاقدره لي ويسره لي ثم بارك لي فيه وان كنت تعلم ان هذا الامر شر لي في ديني ومعاشي وعاقبة امري فاصرفه عني واصرفني عنه واقدر لي الخير حيث كان ثم ارضني به وفي مسند الامام احمد من حديث سعد بن ابي وقاص عن النبي ص انه قال من سعادة ابن ادم استخارة الله ومن سعادة ابن ادم رضاه بما قضى الله ومن شقوة ابن ادم تركه استخارة الله ومن شقوة ابن ادم سخطه بما قضى الله وقد قال سبحانه وتعالى
 \textmash{]
وشاورهم في الامر فاذا عزمت فتوكل على الله
 }]
وقال قتاده ما تشاور قوم يبتغون وجه الله الا هدوا الى ارشد امرهم
\L{This is a simple example of Arabic text you may want to type}
تم والحمد لله رب العالمين.
\end{document}
```

Figure 2.7: Sample Arabi input

بسم الله الرحمن الرحيم ،
الفصل السادس عشر في الاستخارة

في صحيح البخاري عن جابر قال كان رسول الله يعلمنا الاستخارة في الأمر كما يعلمنا السورة من القران إذا هم أحدكم بالأمر فليركع ركعتين من غير الفريضة ثم ليقل اللهم إني أستخيرك بعلمك وأستقدرك بقدرتك وأسألك من فضلك العظيم فانك تقدر ولا اقدر وتعلم ولا اعلم وأنت علام الغيوب اللهم إن كنت تعلم أن هذا الأمر ويسمى حاجته خير لي في ديني ومعاشي وعاقبة أمري فاقدره لي ويسره لي ثم بارك لي فيه وان كنت تعلم أن هذا الأمر شر لي في ديني ومعاشي وعاقبة أمري فاصرفه عني واصرفني عنه واقدر لي الخير حيث كان ثم أرضني به وفي مسند الإمام احمد من حديث سعد بن أبي وقاص عن النبي ص انه قال من سعادة ابن ادم استخارة الله ومن سعادة ابن ادم رضاه بما قضى الله ومن شقوة ابن ادم تركه استخارة الله ومن شقوة ابن ادم سخطه بما قضى الله وقد قال سبحانه وتعالى ﴿وشاورهم في الأمر فإذا عزمت فتوكل على الله﴾ وقال قتادة ما تشاور قوم يبتغون وجه الله إلا هدوا إلى ارشد أمرهم This is a simple example of Arabic text you may want to type تم والحمد لله رب العالمين.

Figure 2.8: Sample Arabi output

2.5 Amiri Font

Amiri is a classical Arabic typeface in Naskh style for typesetting books and other running text. [14]

Amiri according to its developer description is a revival of the beautiful typeface pioneered in early 20th century by Bulaq Press in Cairo, also known as Amiria Press, after which the font is named.

The uniqueness of this typeface comes from its superb balance between the beauty of Naskh calligraphy on one hand, the constraints and requirements of elegant typography on the other. Also, it is one of the few metal typefaces that were used in typesetting the Koran, making it a good source for a digital typeface to be used in typesetting Koranic verses.

Amiri project aims at the revival of the aesthetics and traditions of Arabic typesetting, and adapting it to the era of digital typesetting, in a publicly available form.

Amiri is a free, open source project that everyone can use and modify.

Figures 2.9 and 2.10 show the examples of Amiri font.

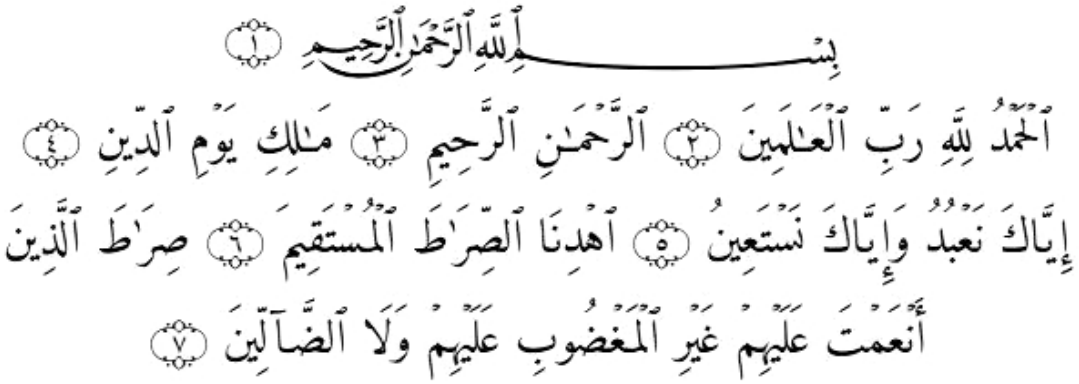


Figure 2.9: Surat Al-Fatiha set in Amiri

These examples can be reproduced by downloading the Amiri font package and using the TrueType font files (*.ttf) with Microsoft Word or other editors that supports TrueType.

Figure 2.10: Sample of Amiri fonts

2.6 Microsoft Word

Microsoft Office supports right-to-left functionality and features for languages that work in a right-to-left or a combined right-to-left, left-to-right environment for entering, editing, and displaying text. In this context, "right-to-left languages" refers to any writing system that is written from right to left and includes languages that require contextual shaping, such as Arabic, and languages that do not. You can change your display to read right-to-left or change individual files so their contents read from right to left.

If your computer doesn't have a right-to-left language version of Office installed, you will need to install the appropriate language pack. You must also be running a Microsoft Windows operating system that has right-to-left support and enable the keyboard language for the right-to-left language that you wants to use.

In word-processing programs — such as Word — you can select, find, and replace individual diacritics and individual Arabic characters regardless of whether they are ligated. Each ligature and diacritic is managed as a discrete unit of a right-to-left language word.

Figure 2.11 shows selecting an Arabic word that has a three-character ligature (as each character is selected).

When it comes to Mathematical expressions, despite MS Office's ease of use (as it is based on WYSIWYG concept) unlike the previous techniques, The Equation Editor cheat sheet provides an impressive list of symbols, but it doesn't come close to the amount of symbols available in LaTeX . The Comprehensive LaTeX Symbol List has 100 pages of symbols.

For fonts, it offers a set of Arabic fonts that are somewhat not flexible to represent the

Figure 2.11: Selecting an Arabic word that has a three-character ligature in MS Word

beauty and versatility of Arabic writings. But it supports adding custom OpenType and TrueType fonts.

Here is a sample of fonts supplied with Windows 8 and targeting Arabic script:
Aldhabi: Shown in figure 2.12
Andalus: Shown in figure 2.13
Arabic Typesetting: Shown in figure 2.14
Microsoft Uighur: Shown in figure 2.15
Sakkal Majalla: Shown in figure 2.16
Simplified Arabic: Shown in figure 2.17
Traditional Arabic: Shown in figure 2.18
Urdu Typesetting: Shown in figure 2.19

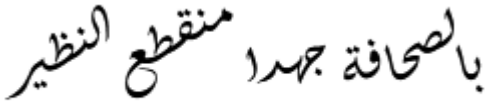


Figure 2.12: Aldahabi font

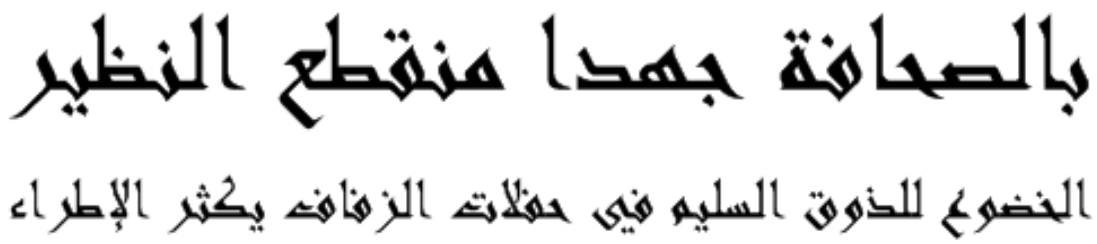


Figure 2.13: Andalus font

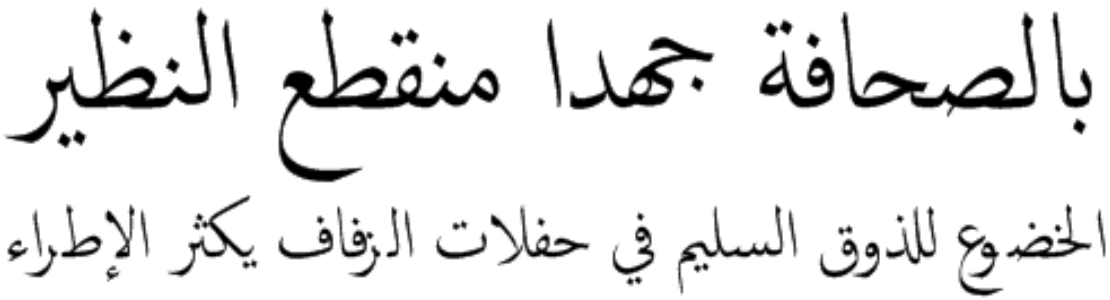


Figure 2.14: Arabic Typesetting font

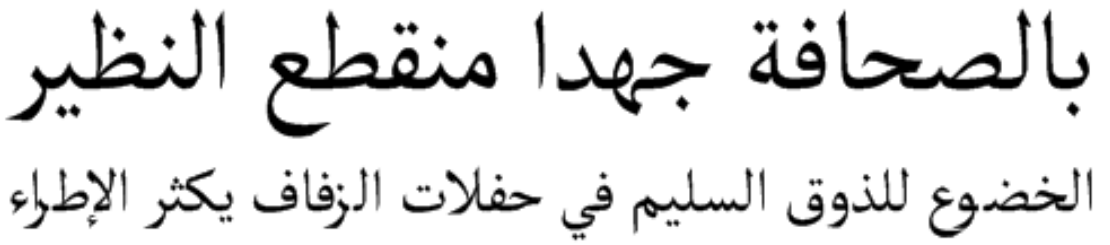


Figure 2.15: Microsoft Uighur font

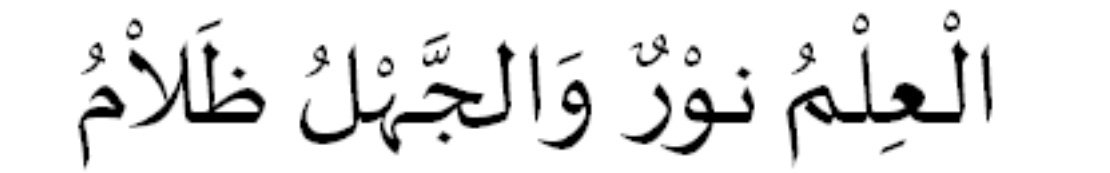


Figure 2.16: Sakkal Majalla font

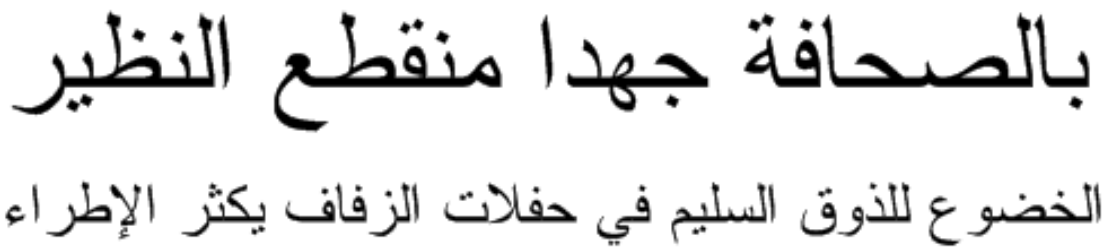


Figure 2.17: Simplified Arabic font

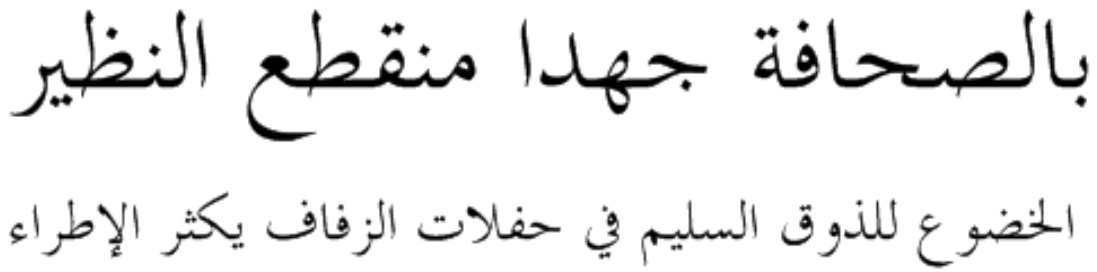


Figure 2.18: Traditional Arabic font

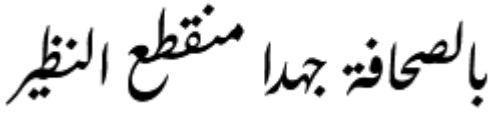


Figure 2.19: Urdu Typesetting font

2.7 Hand Written Printings

For a long time, most famous Arabic educational mathematics books were written by hand. There weren't any tool that had the ability to provide the required large set of mathematical symbols and help typesetting the expressions used in the mathematical problems and their answers for the different academic years. So, the editors of these books had to manually typeset them.

Examples: Almo'aser in mathematics is the most famous book for mathematical problems and their solutions for secondary education. Here are some examples from a 1999 printing of the book. [15] in figures 2.20 and 2.21:

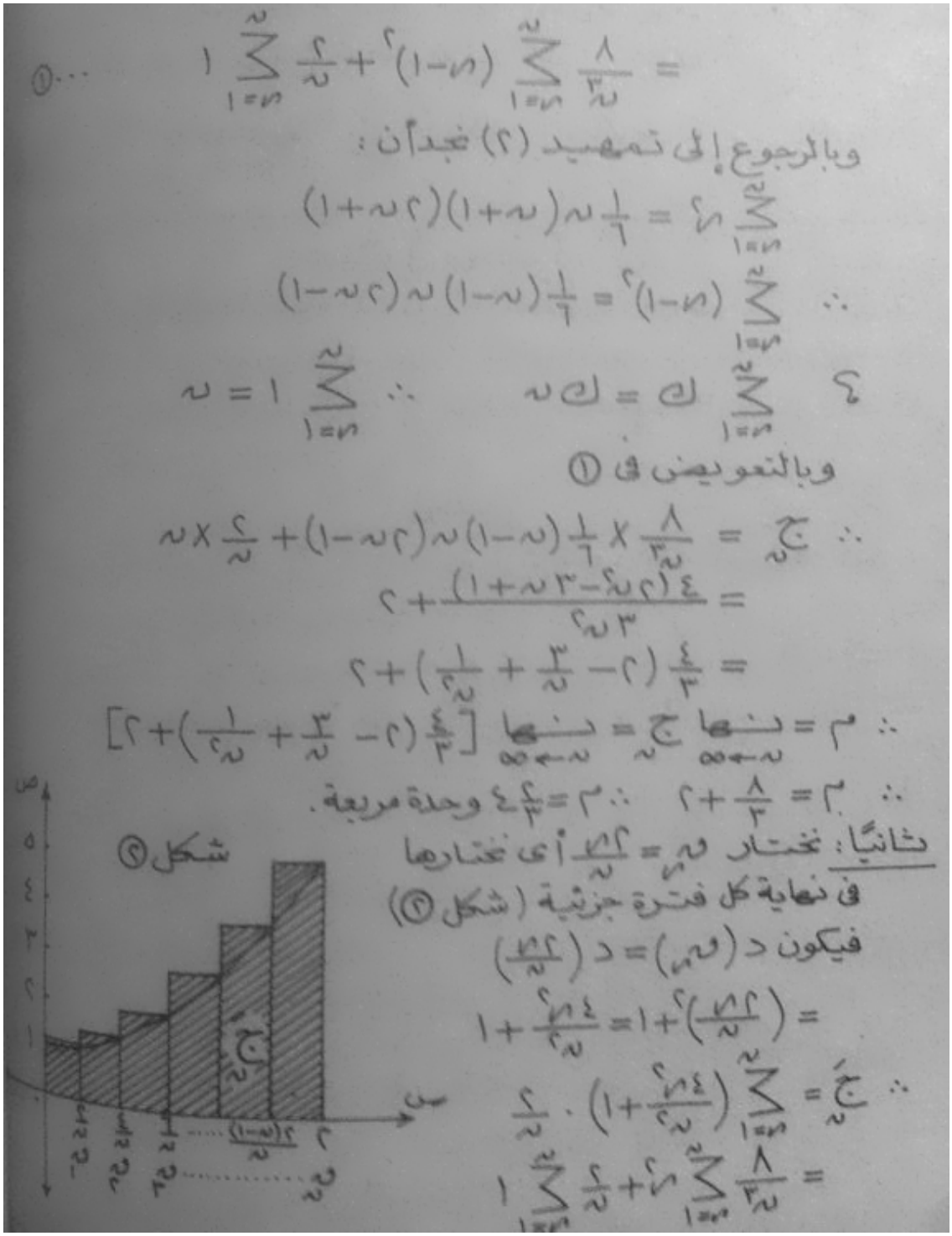


Figure 2.20: Example 1 from AlMo'aser Book for secondary education

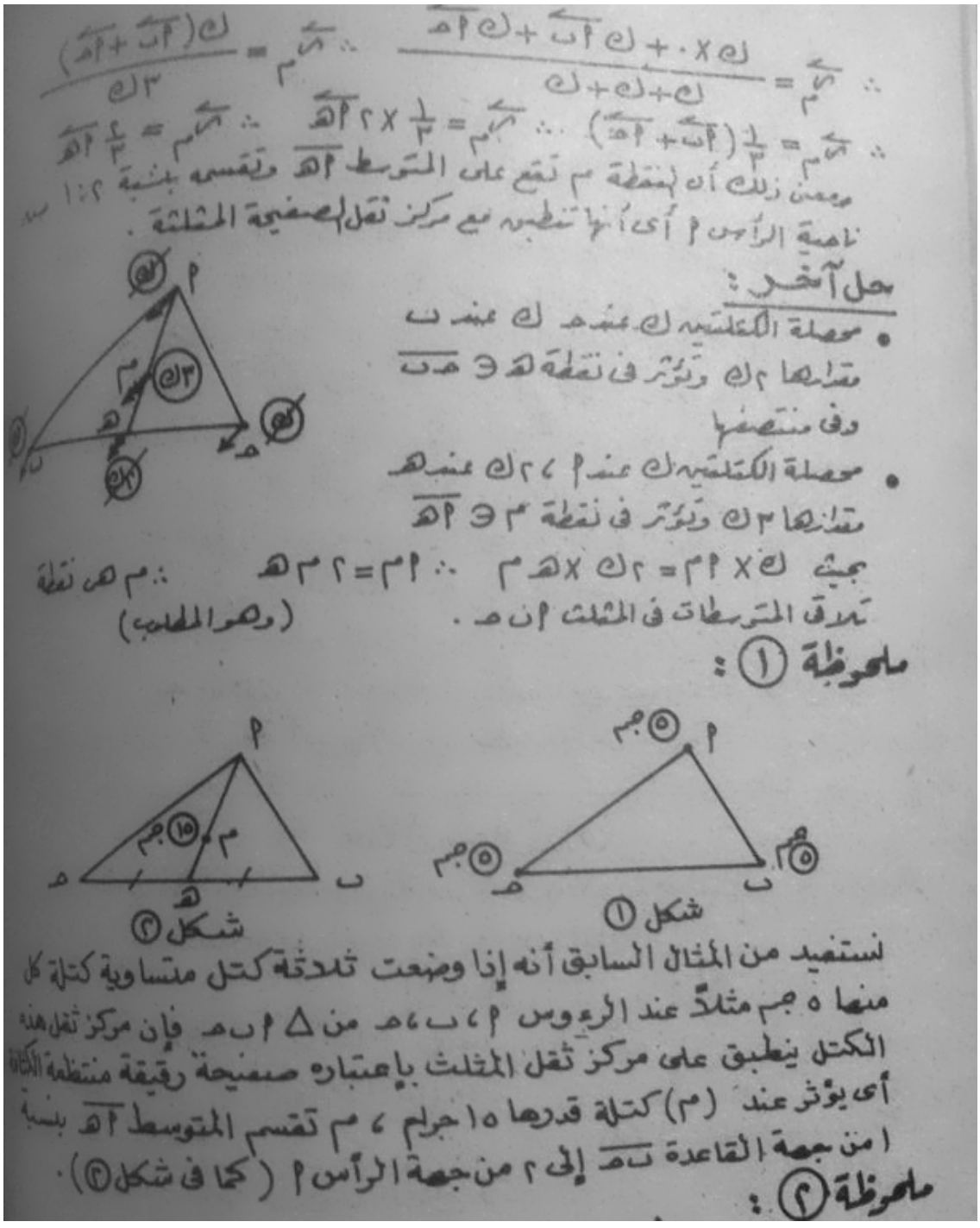


Figure 2.21: Example 2 from AlMo'aser Book for secondary education

Chapter 3

Font Development Overview

Here we will talk generally about required font development tools and the used algorithms for building a complex and rich font like AlQalam:

3.1 TEX

TEX (= tau epsilon chi) is a computer language designed for use in typesetting; in particular, for typesetting math and other technical (from Greek "techne" = art/craft, the stem of `technology') material.

In the late 1970s, Donald Knuth was revising the second volume of his multivolume magnum opus The Art of Computer Programming, got the galleys, looked at them, and said (approximately) "blecch"! He had just received his first samples from the new typesetting system of the publisher's, and its quality was so far below that of the first edition of Volume 2 that he couldn't stand it. Around the same time, he saw a new book (Artificial Intelligence, by Patrick Winston) that had been produced digitally, and ultimately realized that typesetting meant arranging 0's and 1's (ink and no ink) in the proper pattern, and said (approximately), "As a computer scientist, I really identify with patterns of 0's and 1's; I ought to be able to do something about this", so he set out to learn what were the traditional rules for typesetting math, what constituted good typography, and (because the fonts of symbols that he needed really didn't exist) as much as he could about type design. He figured this would take about 6 months. (Ultimately, it took nearly 10 years, but along the way he had lots of help from some people who are well known to many readers here—Hermann Zapf, Chuck Bigelow, Kris Holmes, Matthew Carter and Richard Southall are acknowledged in the introduction to Volume E, Computer Modern Typefaces, of the Addison-Wesley Computers and Typesetting book series.)

A year or so after he started, Knuth was invited by the American Mathematical Society (AMS) to present one of the principal invited lectures at their annual meeting. This honor is awarded to significant academic researchers who (mostly) were trained as mathematicians, but who have done most of their work in not strictly mathematical areas (there are a number of physicists, astronomers, etc., in the annals of this lecture series as well as computer scientists); the lecturer can speak on any topic s/he wishes, and Knuth decided to speak on computer science in the service of mathematics. The topic he presented was his new work on TEX (for typesetting) and Metafont (for developing fonts for use with TEX). He presented not only the roots of the typographical concepts, but also the mathematical notions (e.g., the use of Bezier splines to shape glyphs) on which these two programs are based. The programs sounded like they were just about ready to use, and

quite a few mathematicians, including the chair of the Math Society's board of trustees, decided to take a closer look. As it turned out, TEX was still a lot closer to a research project than to an industrial strength product, but there were certain attractive features:

it was intended to be used directly by authors (and their secretaries) who are the ones who really know what they are writing about; it came from an academic source, and was intended to be available for no monetary fee (nobody said anything about how much support it was going to need); as things developed, it became available on just about any computer and operating system, and was designed specifically so that input files (files containing markup instructions; this is not a WYSIWYG system) would be portable, and would generate the same output on any system on which they were processed—same hyphenations, same line breaks, same page breaks, same everything. other programs available at the time for mathematical composition were: proprietary; very expensive often limited to specific hardware, if WYSIWYG, the same expression in two places in the same document might very well not look the same, never mind look the same if processed on two different systems. Mathematicians are traditionally, shall we say, frugal; their budgets have not been large (before computer algebra systems, pencils, paper, chalk and blackboards were the most important research tools). TEX came along just before the beginnings of the personal computer; although it was developed on one of the last of the "academic" mainframes (the DECsystem ("Edusystem")-10 and -20), it was very quickly ported to some early HP workstations and, as they emerged, the new personal systems. From the start, it has been popular among mathematicians, physicists, astrophysicists, astronomers, any research scientists who were plagued by lack of the necessary symbols on typewriters and who wanted a more professional look to their preprints.

To produce his own books, Knuth had to tackle all the paraphernalia of academic publishing—footnotes, floating insertions (figures and tables), etc., etc. As a mathematician/computer scientist, he developed an input language that makes sense to other scientists, and for math expressions, is quite similar to how one mathematician would recite a string of notation to another on the telephone. The TEX language is an interpreter. It accepts mixed commands and data. The command language is very low level (skip so much space, change to font X, set this string of words in paragraph form, …), but is amenable to being enhanced by defining macro commands to build a very high level user interface (this is the title, this is the author, use them to set a title page according to AMS specifications). The handling of footnotes and similar structures are so well behaved that "style files" have been created for TEX to process critical editions and legal tomes. It is also (after some highly useful enhancements in about 1990) able to handle the composition of many different languages according to their own traditional rules, and is for this reason (as well as for the low cost), quite widely used in eastern Europe and with other scripts outside of western Europe/North America.

Some of the algorithms in TEX have not been bettered in any of the composition tools devised in the years since TEX appeared. The most obvious example is the paragraph breaking: text is considered a full paragraph at a time, not line-by-line; this is the basic starting algorithm used in the HZ-program by Peter Karow (and named for Hermann Zapf, who developed the special fonts this program needs to improve on the basics).

A TEX system stands on its own, provided all the fonts one needs are available. TEX uses only the metrics, and produces a "device independent" output file—.dvi—that must then be translated to the particular output device being used (an imagesetter, laser printer, inkjet printer; in the "old days" even daisy-wheel printers were used). The DVI translator actually accesses the font shapes, either as bitmaps, Type 1 fonts, or pointers to fonts

installed in a printer with the shapes not otherwise accessible. PostScript and PDF are two of the most popular "final" output forms for TEX .

One of the major areas where TEX holds its own is as a "back end" to SGML and XML systems, where no human intervention is expected between data input (structured, not WYSIWYG) and removing the output from the printer or viewing it on a screen. Granted, this isn't "creative" in the sense most often discussed, but it's still important to the readability and usefulness of such documents that care is taken with the design and typography, and the flexibility and programmability of TEX makes that possible.

In summary, TEX is a special-purpose programming language that is the centerpiece of a typesetting system that produces publication quality mathematics (and surrounding text), available to and usable by individuals.

3.2 Metafont

Metafont is a program for making bitmap fonts for use by TEX , its viewers, printer drivers, and related programs. It interprets a drawing language with a syntax apparently derived in part from the Algol2 family of programming languages, of which C, C++, Pascal and Modula-2 are members. The input can be interactive, or from a source file. Metafont source files are usually suffixed '.mf'. Metafont sources can utilize scaling, rotation, reflection, skewing and shifting, and other complex transformations in obvious and intuitive ways. But that is another story, told (in part) by The Metafont book. Metafont 's bitmap output is a gf (generic font) file. This may be compressed to an equivalent pk (packed) font by the auxiliary program GFtoPK. Why doesn't Metafont output pk fonts directly? Firstly, Tomas Rokicki had not invented pk at the time Donald E. Knuth was writing Metafont . Secondly, to change Metafont now would be too big a change in Knuth's opinion. (Knuth is a very conservative programmer; this fact is a two-sided coin.) gf and pk files are suffixed '.*gf' and '.*pk' respectively, where, in a typical unix installation, the '*' stands for the font resolution. (Resolution will be explained below.)

A bitmap is all that's needed for large-scale proofs, as produced by the GFtoDVI utility, but for TEX to typeset a font it needs a tfm (TEX Font Metric) file to describe the dimensions, ligatures and kerns of the font. Metafont can be told to make a tfm file, by making the internal variable 'fontmaking' positive.

Remember that TEX reads only the tfm files. The glyphs, or forms of the characters, as stored in gf or pk font files, do not enter the picture (I mean, are not read) until the dvi drivers are run.

TEX can scale tfm files. Unfortunately, bitmaps such as gf and pk are not scalable. However, Metafont files can be compiled into fonts of arbitrary scale by Metafont , even by non-programmers. Incidentally, properly constructed tfm files are device-independent, so running Metafont with different modes normally produces the identical tfm. Dimensions in tfm files are specified to Metafont in device independent 'sharped' dimensions (commonly suffixed by #), where a value of 1 corresponds to the dimension of 1pt (typographical point). Most of Metafont 's calculations are done with (resolution and device dependent) pixels as units. Care must be taken by font designers to always calculate unsharped dimensions from sharped ones, and never the other way round, so as to keep roundoff errors or similar effects from influencing the tfm files to depend on resolution or device. Although type quality will be influenced only in minuscule ways, this is one of the more common reasons for checksum errors reported by printer drivers. Note that the

only way to be sure that a TFM file is device-independent is to create the font in different modes and compare the resulting TFM's, perhaps using tftopl. More detailed descriptions of tfm and gf files, and of proof mode, are found in Appendices F, G, and H, respectively of The Metafont book.

Metafont, Knuth's font creation program, is independent of TEX . It generates only bitmap fonts (although internally it creates outlines on which the bitmaps are based). There is still research to be done on combining overlapping outlines into a single outline that can be used like the outline of a Type 1 font; Knuth has "frozen" Metafont, so further research and development are being done by someone else, and the result will not be called "Metafont". In fact, it's also possible to use Type 1 fonts with TEX , and nearly all TEX installations routinely use free or commercial Type 1 fonts, especially if they're not producing heavily technical material; only Computer Modern (the font Knuth developed), Lucida Bright, and Times have anywhere near a comprehensive symbol complement, and none of the symbol sets are from "major" font suppliers (too much work, not enough money; the demise of Monotype's symbol catalog—metal only—is a particularly great loss).

TEX is the composition engine (strictly speaking, an interpreter, not a compiler). It is essentially a batch engine, although a limited amount of interactivity is possible when processing a file, to allow error recovery and diagnosis. Thus it *is* a page layout application.

3.3 MetaPost

MetaPost is a programming language much like Knuth's Metafont except that it outputs vector graphics, either PostScript programs or SVG graphics, instead of bitmaps. Borrowed from Metafont are the basic tools for creating and manipulating pictures. These include numbers, coordinate pairs, cubic splines, affine transformations, text strings, and boolean quantities. Additional features facilitate integrating text and graphics and accessing special features of PostScript such as clipping, shading, and dashed lines. Another feature borrowed from Metafont is the ability to solve linear equations that are given implicitly, thus allowing many programs to be written in a largely declarative style. By building complex operations from simpler ones, MetaPost achieves both power and flexibility. MetaPost is particularly well-suited to generating figures for technical documents where some aspects of a picture may be controlled by mathematical or geometrical constraints that are best expressed symbolically. In other words, MetaPost is not meant to take the place of a freehand drawing tool or even an interactive graphics editor. It is really a programming language for generating graphics, especially figures for TEX and troff documents. MetaPost language is based on Knuth's Metafont to a large extent.

3.4 LuaTEX

LuaTEX is an extended version of pdfTEX using Lua as an embedded scripting language. The LuaTEX project's main objective is to provide an open and configurable variant of TEX while at the same time offering downward compatibility. From the user perspective we have pdfTEX as stable and more or less frozen 8 bit engine, XeTEX as unicode input and font aware engine using libraries for font handling, and LuaTEX as engine that is

programmable and delegates as much as possible to Lua, with the objective to keep the core engine lean and mean.

3.5 Lua Language

Lua is a powerful, fast, lightweight, embeddable scripting language.

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

Lua has been used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems (e.g., the Ginga middleware for digital TV in Brazil) and games (e.g., World of Warcraft and Angry Birds). Lua is currently the leading scripting language in games. Lua has a solid reference manual and there are several books about it. Several versions of Lua have been released and used in real applications since its creation in 1993. Lua featured in HOPL III, the Third ACM SIGPLAN History of Programming Languages Conference, in June 2007. Lua won the Front Line Award 2011 from the Game Developers Magazine.

It has a deserved reputation for performance. To claim to be "as fast as Lua" is an aspiration of other scripting languages. Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages. Lua is fast not only in fine-tuned benchmark programs, but in real life too. Substantial fractions of large applications have been written in Lua.

Lua is distributed in a small package and builds out-of-the-box in all platforms that have a standard C compiler. Lua runs on all flavors of Unix and Windows, on mobile devices (running Android, iOS, BREW, Symbian, Windows Phone), on embedded microprocessors (such as ARM and Rabbit, for applications like Lego MindStorms), on IBM mainframes, etc.

Lua is a fast language engine with small footprint that you can embed easily into your application. Lua has a simple and well documented API that allows strong integration with code written in other languages. It is easy to extend Lua with libraries written in other languages. It is also easy to extend programs written in other languages with Lua. Lua has been used to extend programs written not only in C and C++, but also in Java, C#, Smalltalk, Fortran, Ada, Erlang, and even in other scripting languages, such as Perl and Ruby.

A fundamental concept in the design of Lua is to provide meta-mechanisms for implementing features, instead of providing a host of features directly in the language. For example, although Lua is not a pure object-oriented language, it does provide meta-mechanisms for implementing classes and inheritance. Lua's meta-mechanisms bring an economy of concepts and keep the language small, while allowing the semantics to be extended in unconventional ways.

Adding Lua to an application does not bloat it. The tarball for Lua 5.3.0, which contains source code and documentation, takes 272K compressed and 1.1M uncompressed. The source contains around 23000 lines of C. Under 64-bit Linux, the Lua interpreter built with all standard Lua libraries takes 241K and the Lua library takes 412K.

Lua is free open-source software, distributed under a very liberal license (the well-

known MIT license). It may be used for any purpose, including commercial purposes, at absolutely no cost. Just download it and use it. [16]

3.6 Context Analysis

By "contextual analysis" we mean the determination of a character's proper presentation form according to its context.

In Arabic each character may have several presentation forms. The proper presentation form of a character in a text is determined according to the available presentation forms of the character itself and those of characters surrounding it. It also depends on the current presentation forms of the surrounding characters. Determining the proper presentation form of a character according to its context is called "contextual analysis". [17]

3.7 Line Breaking in TEX

Line Breaking is the mechanism for choosing the "best possible" breakpoints that yield the individual lines of a paragraph.

The algorithm studied and discussed here is based on an approach devised by Michael F. Plass and D. E. Knuth in 1977, subsequently generalized and improved by the same two people in 1980 for TEX .

A detailed discussion appears in [18].

3.7.1 Algorithm Overview

- TEX 's line-breaking algorithm takes a given horizontal list (unprocessed paragraph) and converts it to a sequence of boxes that are appended to the current vertical list (page lines).
- While doing this, it creates a special data structure containing three kinds of records that are not used elsewhere in TEX which make the other parts of TEX do not need to know anything about how line-breaking is done.
- General outline of line_ break procedure (the main lengthy procedure that handles line breaking in TEX):
  - Get ready to start line breaking.
  - Find optimal breakpoints.
  - Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and append them to the current vertical list.
  - Clean up the memory by removing the break nodes.

3.7.2 Break Nodes

- TEX creates a “break node” for each break that is “feasible”, in the sense that there is a way to end a line at the given place without requiring any line to stretch more than a given tolerance.

- A break node is characterized by three things:
- Position of the break (a pointer to a glue_ node, math_ node, penalty_ node, or disc_ node).
- Ordinal number of the line that will follow this breakpoint.
- The fitness classification of the line that has just ended, i.e., tight_ fit, decent_ fit, loose_ fit, or very_ loose_ fit.
- Fitness classifications:
- tight_ fit=0 lines shrinking 0.5 to 1.0 of their shrinkability
- loose_ fit=2 lines stretching 0.5 to 1.0 of their stretchability
- very_ loose_ fit=3 lines stretching more than their stretchability
- decent_ fit=1 all other lines
- The algorithm essentially determines the best possible way to achieve each feasible combination of position, line, and fitness.

3.7.3 Active and Passive Nodes

- An "active node" and a "passive node" are created in memory for each feasible breakpoint that needs to be considered.
- We need active nodes only for breakpoints near the place in the paragraph that is currently being examined, so they are recycled within a comparatively short time after they are created.
- An active node for a given breakpoint contains six fields:
- link: points to the next node in the list of active nodes.
- break_ node: points to the passive node associated with this breakpoint.
- line_ number: is the number of the line that begins at this breakpoint.
- fitness: is the fitness classification of the line ending at this breakpoint.
- type: is either hyphenated(1) or unhyphenated(0), depending on whether this breakpoint is a disc_ node.
- total_ demerits: is the minimum possible sum of demerits over all lines leading from the beginning of the paragraph to this breakpoint (the quantity that TEX minimizes).
- The passive node for a given breakpoint contains only four fields:
- link: points to the passive node created just before this one, if any, otherwise it is null.
- cur_ break: points to the position of this breakpoint in the horizontal list for the paragraph being broken.

- prev_ break: points to the passive node that should precede this one in an optimal path to this breakpoint.
- serial: is equal to n if this passive node is the nth one created during the current pass. (This field is used only when printing out detailed statistics about the line-breaking calculations.)
- The active list also contains “delta "nodes that help the algorithm compute the badness of individual lines.
- Such nodes appear only between two active nodes, and they have type=delta_ node(2).
- If p and r are active nodes and if q is a delta node between them, so that link(p)=q and link(q)=r, then q tells the space difference between lines in the horizontal list that start after breakpoint p and lines that start after breakpoint r. Delta Nodes:
- The active list also contains “delta "nodes that help the algorithm compute the badness of individual lines.
- Such nodes appear only between two active nodes, and they have type=delta_ node(2).
- If p and r are active nodes and if q is a delta node between them, so that link(p)=q and link(q)=r, then q tells the space difference between lines in the horizontal list that start after breakpoint p and lines that start after breakpoint r.
- In other words, if we know the length of the line that starts after p and ends at our current position, then the corresponding length of the line that starts after r is obtained by adding the amounts in node q.
- A delta node contains six scaled numbers, since it must record the net change in glue stretchability with respect to all orders of infinity.
- The six scaled numbers are: natural width difference , the stretch differences in units of pt, fil, fill, and filll and the shrink difference.
- Glue nodes in a horizontal list that is being paragraphed are not supposed to include “infinite‘” shrinkability; that is why the algorithm maintains four registers for stretching but only one for shrinking.

3.7.4 Finding Feasible Breakpoints

- A global pointer variable cur_ p (used both by line_ break and by its subprocedure try_ break) runs through the given hlist as we look for breakpoints.
- Another global variable called threshold is used to determine the feasibility of individual lines:
- Breakpoints are feasible if there is a way to reach them without creating lines whose badness exceeds threshold.
- The badness(?) is compared to threshold before penalties are added, so that penalty values do not affect the feasibility of breakpoints, except that no break is allowed when the penalty is 10000 or more.

- Up to three passes might be made through the paragraph in an attempt to find at least one set of feasible breakpoints.
- On the first pass, we have threshold=pretolerance.
- If this pass fails to find a feasible solution, threshold is set to tolerance and an attempt is made to hyphenate as many words as possible.
- If that fails too, we add emergency_ stretch to the background stretchability and set final_ pass=true.
- The heart of the line-breaking procedure is 'try_ break', a subroutine that tests if the current breakpoint cur_ p is feasible, by running through the active list to see what lines of text can be made from active nodes to cur_ p.
- If feasible breaks are possible, new break nodes are created.
- If cur_ p is too far from an active node, that node is deactivated.
- The parameter pi to try_ break is the penalty associated with a break at cur_ p. Finding optimal break points:
- As we consider various ways to end a line at cur_ p, in a given line number class, we keep track of the best total demerits known, in an array with one entry for each of the fitness classifications.
- For example:
- minimal_ demerits[tight_ fit] contains the fewest total demerits of feasible line breaks ending at cur_ p with a tight_ fit line
- best_ place[tight_ fit] points to the passive node for the break before cur_ p that achieves such an optimum.
- best_ pl_ line[tight_ fit] is the line_ number field in the active node corresponding to best_ place[tight_ fit].
- When no feasible break sequence is known, the minimal_ demerits entries will be equal to awful_ bad.
- Active nodes with minimal_ demerits greater than minimum_ demerits+abs(adj_ demerits) will never be chosen in the final paragraph breaks.
- This observation allows us to omit a substantial number of feasible breakpoints from further consideration.
- The remaining part of try_ break deals with the calculation of demerits for a break from node r to cur_ p.
- The first thing to do is calculating the badness, b. This value will always be between zero and inf_ bad+1; the latter value occurs only in the case of lines from r to cur_ p that cannot shrink enough to fit the necessary width.

- If the line from r to cur_ p is feasible, its badness is b, and its fitness classification is fit_ class. We will compute the total demerits and record them in the minimal_ demerits array, and check if such a break is the current champion among all ways to get to cur_ p in a given line-number class and fitness class.
- At last, we find active nodes with fewest demerits for each line class.
- Once the best sequence of breakpoints has been found, we call on the procedure post_ line_ break to finish the remainder of the work:
- Break the paragraph at the chosen breakpoints.
- Justify the resulting lines to the correct widths.
- Append them to the current vertical list.
- By introducing this subprocedure, we are able to keep line_ break from getting extremely long.

Chapter 4

Font Package Development and QC

In this chapter we will talk about the starting point of the project which was creating a usable font package. This step leaded us to begin using the font inside TEX environment and hence discover some bugs in the package like missing forms and joint problems and fix it.

4.1 Creating the Font Package

Metafont is a system that allows you to create fonts easily. Metafont is actually a single command-line application (called mf on most platforms, and mf.exe on Windows platforms). In short, it has no graphical interface. It must be called from a command line or by a helper program. So, how can you create fonts with such a program if you cannot draw figures and such? This program is an interpreter. It takes a series of instructions as input, and executes them one at a time, as it receives them. In other words, Metafont is also a programming language, like C, BASIC, Pascal (in which the source of Metafont was originally written), Perl, and of course TEX . So, how do you enter your programs so that Metafont can interpret them? Well, there are various ways, but the most common is to make a file containing the instructions, and to give the file's name to mf. The file itself must be plain text, and must have the extension .mf. mf will then read this file and give back (if everything's okay) two other files with the same name but different extensions (the process is called compilation). One file will have the extension .tfm. It's called the font metrics file. It contains information like the size of the characters. The second file will have an extension of the form .<number>gf where <number> is a certain value with usually three figures. It contains the actual shapes of the characters (or glyphs, as they are usually called). Both files form the font, and both are necessary to have a working font. So, is that it? Well, it would be too simple if it was. For historical reasons, the gf file is not the one which is actually used. You need first to transform it into a smaller file, containing the same information but in a packed way. You do it with a second program called GFtoPK (GFtoPK.exe on Windows platforms of course), and the resulting file has the same name as the original one, except that its extension has become .<number>pk (with the same number as the gf file). The pk and tfm files form the actual font, and the last thing you have to do is to put them in a place where they will be recognized by TEX and LATEX . Once done, you can happily use your fonts in LATEX .

So, to summarize the steps we can put them in points:

1. Write one (or more) plain text file containing Metafont instructions (a Metafont

program thus) and save it (or them) with a .mf extension.

2. Call mf on this file. The result will be two files, ending respectively in .tfm and .<number>gf.

3. Pack the gf file by calling GFtoPK on it. You will get another file ending in .<number>pk or .pk.

4. Move your tfm and pk files to some place where they can be recognized by TEX and LATEX .

5. Call the font inside your TEX file and start using it.

In our case, all the Naskh style individual letters were developed using Metafont . [19]

The first step for here was creating a one .mf file for all the letters to generate one package for it. A shell script was developed to handle creating what we will call qalambase.mf file which contain all Metafont definitions of all of our letters.

Here is the shell script code that builds the qalambase.mf file.

```
#!/bin/bash

DATE=`date +%e-%m-%y`
echo "%%This file was created using mf files
of $1 directory on $DATE" > $1/qalambase.mf
for file in $1/*.mf
do
  echo "Adding $file macros to qalambase.mf"
  echo "

%%%%%%%%%%%%%%% $file -macros %%%%%%%%%%%%%%%%%
" >> $1/qalambase.mf
  cat "$file" >> $1/qalambase.mf
done
sed "s/$1\///" $1/qalambase.mf | sed "s/\.mf-/-/" |
sed "s/enddef/enddef\;/" |
sed "s/\;\;/\;/" > $1/tmpbase && mv $1/tmpbase $1/qalambase.mf
dos2unix  $1/qalambase.mf
#End of mergemf.sh file
```

First, we run mergemf.sh on the target directory that contain all the .mf files of the font like this:

mergemf.sh mf_ dir

the file qalambase.mf will get generated inside the directory containing all the mf definitions/macros of the letters.

After the qalambase.mf file is generated we run the mf executable on it to get qalam.tfm and qalam.600gf. Then, we run GF2PK on the gf file to get qalam.600pk. Copying the tfm and pk files to the tex fonts directory lets us to call the character definitions inside the TEX file and see the results.

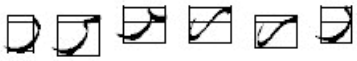


Figure 4.1: Character boxes approach

Figure 4.2: Isolated form of the letter "Noon" without elongation

4.2 Calling the Different Shapes of a Letter

Testing of individual shapes was done along with the development of the font letter forms, form variants, mathematical symbols, etc. Once the basic development was finished, testing of the letters' positions against the baseline and against each other when included in the same word was needed. In addition, there was a need to perform checks for missing forms, to assure smooth joins between letters and correct kerning.

This drove us to call the Metafont letters under TEX to debug the font. To do that, two approaches were possible:

1. Define the character's borders through boxes, similar to the Latin script, and define a character for each shape and with fixed elongation as shown in Fig. 4.1. This approach, by definition, must have a finite number of characters in the font at the end. Such a finite, i.e. limited, number of characters means that we must use only a limited set of values for the parameters of each character.

   So, while debugging we used for example only the "Noon" form shown in Fig. 4.2 despite the font's capability to generate elongated shapes of this form as shown in Fig. 4.3.

2. Using LuaTEX 's embedded Lua and MetaPost engines to call the correct shape with the suitable elongation value. The main advantage of this approach is that we will be able to benefit from all the font features. But lacking experience in this approach made us think that we should postpone using it to the next phases, as more time will be needed to debug it.

Hence we started with the first approach to complete the current debugging phase. The second approach was used later on the project. The next chapter will discuss it in detail.

To surround the resulting letters with boxes, we used the TEX code at Appendix B.1. which produced whats in Fig. 4.4 :

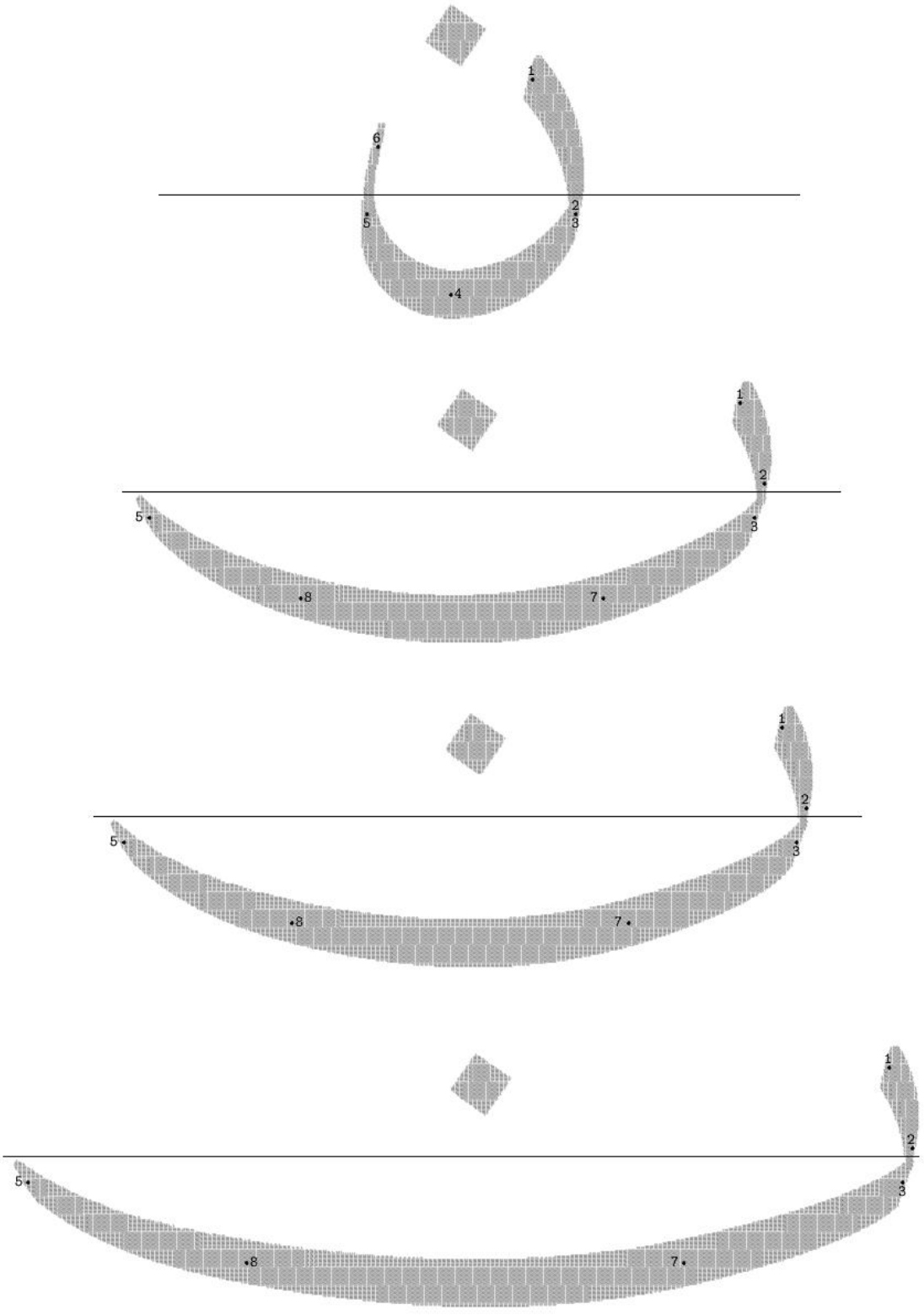


Figure 4.3: Isolated form of the letter "Noon" with different elongation values

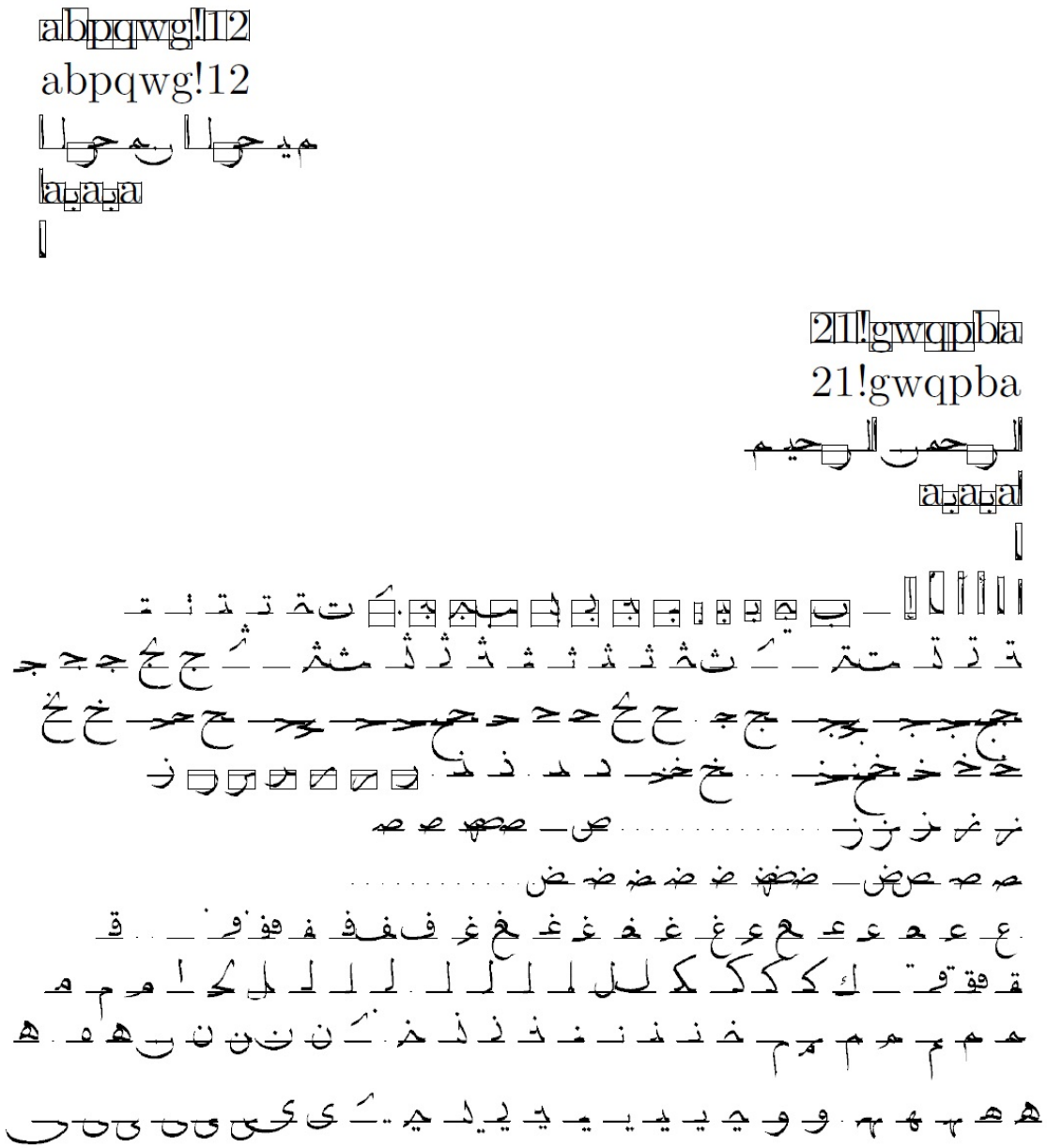


Figure 4.4: Boxes around letters

Figure 4.5: Font state at an early debugging phase

Figure 4.6: Font state after fixing bugs

4.3 Problems Found During Font Debugging

As expected, some bugs appeared when using the font under TEX. If we take a look at Fig. 4.5 we can easily notice the bugs, from right to left as follows. The first and fifth cross signs mark a kerning bug with the letter "Waw" as it appears too far from the succeeding letter. The second and fourth signs mark bad joins of letter "Ain". Also the letter shape itself needs some modifications as marked by the third sign. More bad joins appear between letters "Qaf" and "Sad" and letters "Lam" and "Dal" as marked by the sixth and ninth signs. The seventh sign marks a missing letter that should be added to the font package. The eighth sign marks a mistake in specifying the border of letter "Alef" that caused it to be too close to the succeeding letter.

We have worked on fixing these bugs and many others that appeared in turn, to get to a state with the font that made us ready to move to the next step (despite having very few joining bugs as shown in Fig. 4.6). This is implementing the multi-layer context analysis algorithm to make using the font more user friendly, by automatically choosing the suitable letter forms and form variants instead of the user having to do so manually.

Also, several missing forms of letters were added accordingly as seen in Fig. 4.7

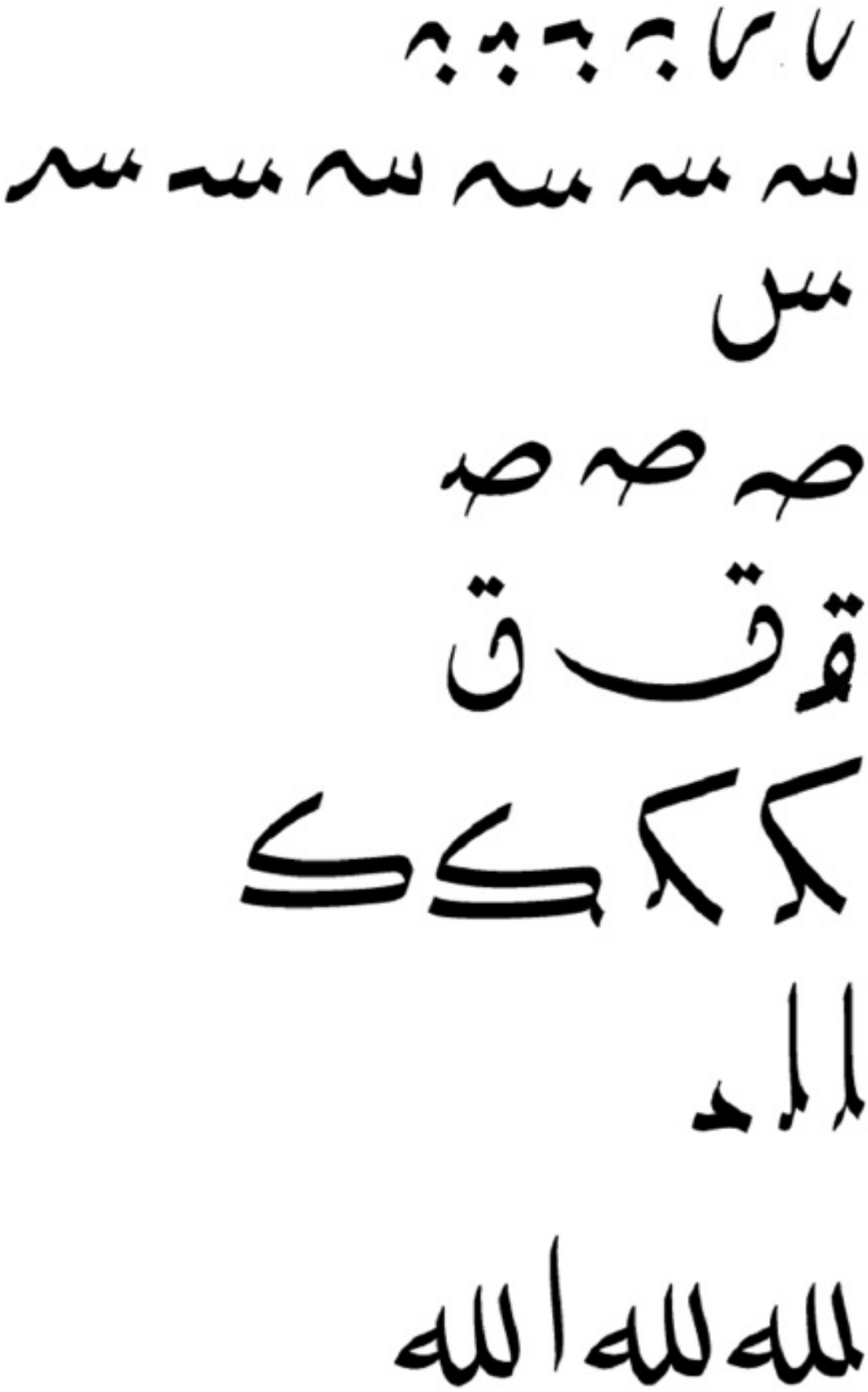


Figure 4.7: Missing forms and glyphs that were added

Chapter 5

Context Analysis Engine Development

In Arabic each character may have several presentation forms. The proper presentation form of a character in a text is determined according to the available presentation forms of the character itself and those of characters surrounding it. It also depends on the current presentation forms of the surrounding characters. Determining the proper presentation form of a character according to its context is called “contextual analysis”. Our target is to develop a generic algorithm that handles context analysis for AlQalam font and similar rich fonts.

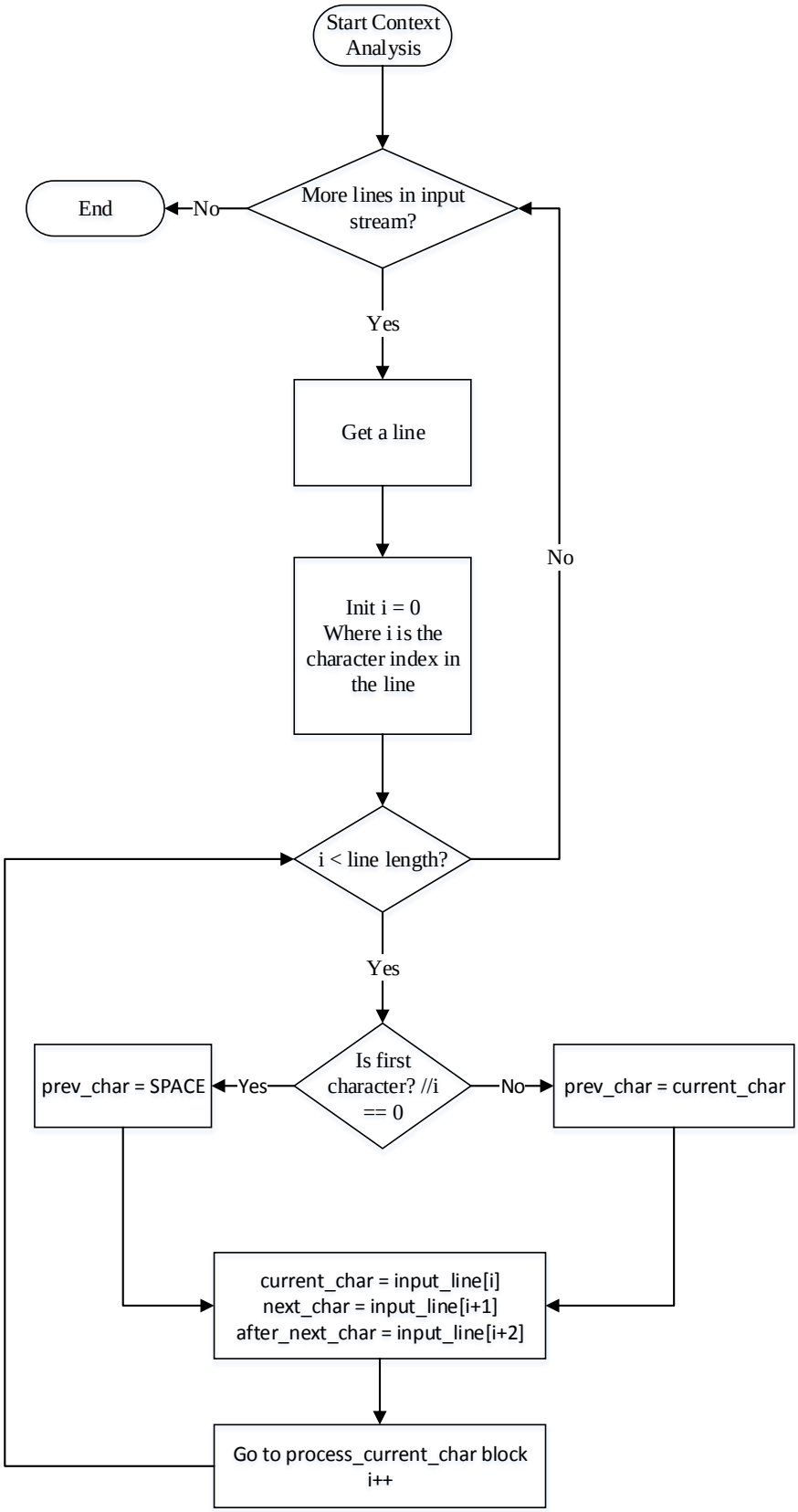
5.1 Context Analysis Engine Using C

Like all multi-lingual computing, Arabic computing is now firmly in the domain of Unicode. Unicode is an industrial protocol with the status of international agreement. It is designed to encode the elements of all known script systems in such a way that they become interchangeable between programs and operating systems.(For more information refer to Arabic standard unicode charts implemented by the Unicode Consortium [20]) As a first approach we have implemented a C program to process an Arabic text file through reading it line by line and each line is read character by character to decide which form should be called for each letter. The output of this third party application was a TEX file that contains function calls to the the letters with the correct forms. This output file should be passed to LaTEX engine to generate a pdf for the text written in AlQalam font with the contextually correct forms.

5.1.1 Algorithm Overview

Referring to (Appendix A) that represents all the letter forms of AlQalam font and the conditions to use each form, we developed a generic algorithm to handle this table and similar kind of font tables.

Here's an top level flow chart for our algorithm:



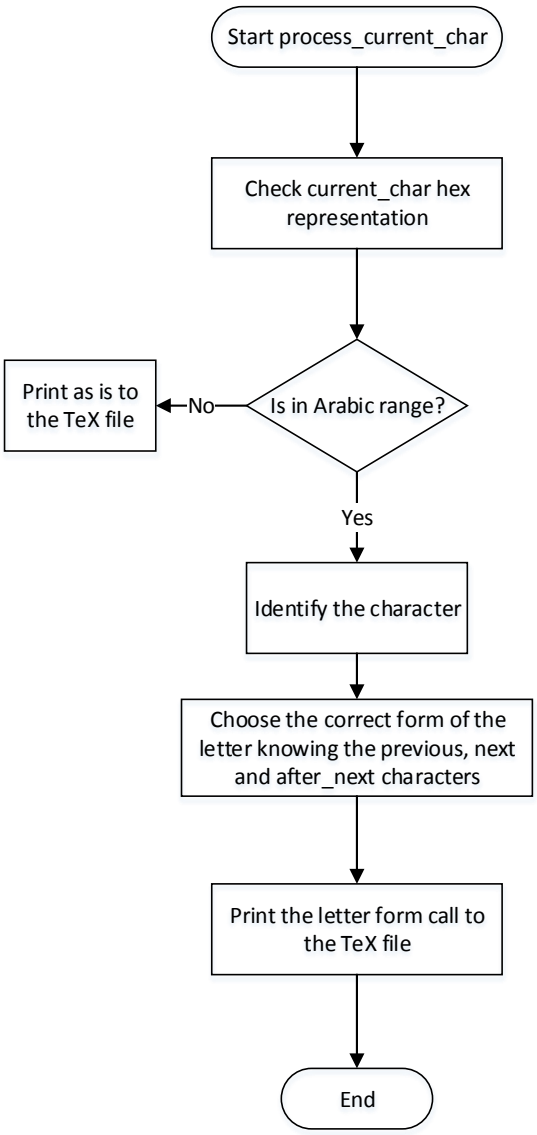
Top Level Flowchart

Let's describe "process_current_char" block in more details: In Unicode, the basic

Arabic characters are in the range of U+0600 to U+0652 and extended Arabic characters are in the range of U+0653 to U+06FF.

Referring to [20], These ranges when represented in the UTF-8 hex are from 0xD880 to 0xD992 for Basic characters and from 0xD993 to 0xDBBF.

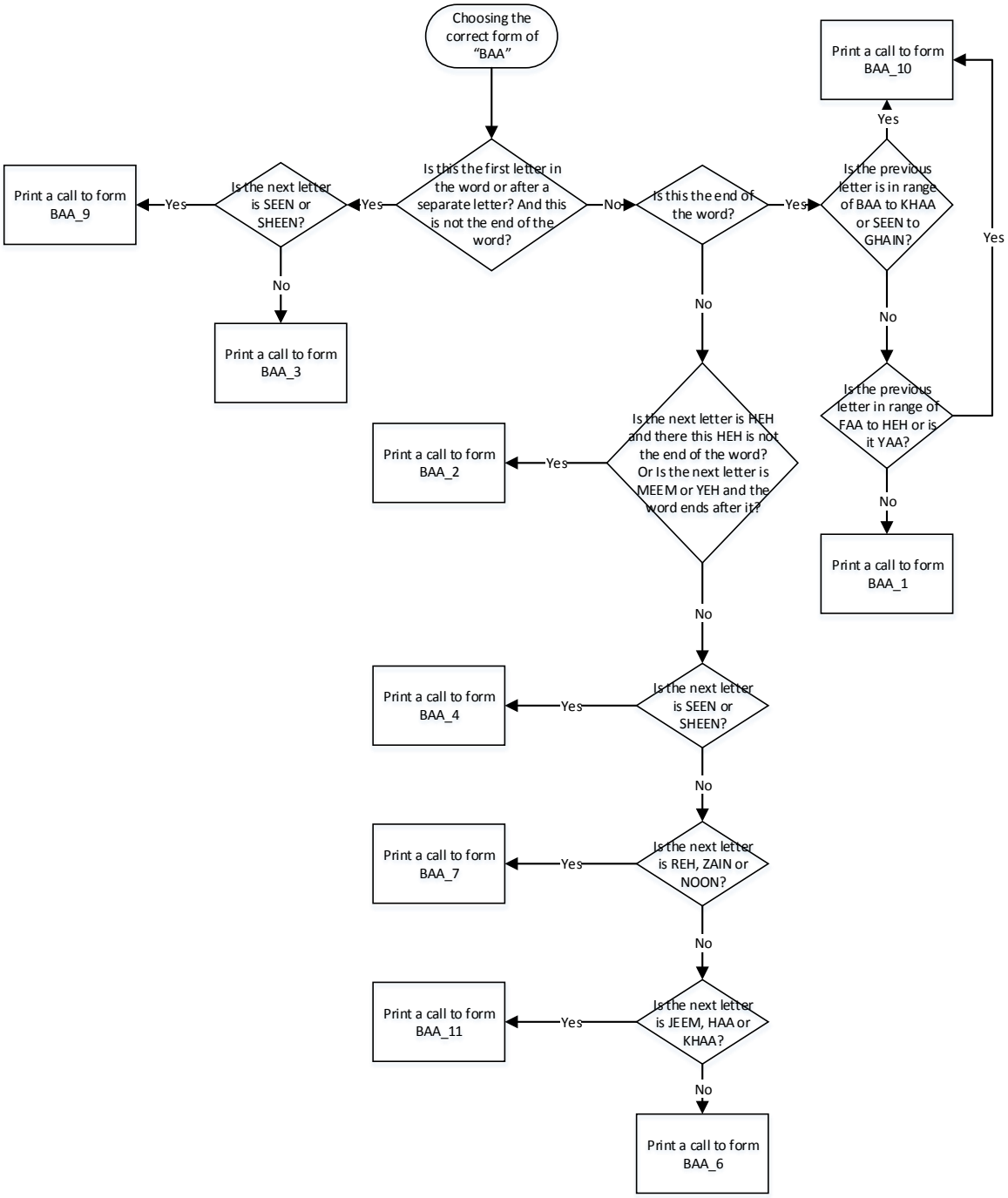
We have covered the 0xD8xx and 0xD9xx range and the coming flowchart describes the behavior of the code according to the UTF-8 hex value of the current character being processed.



process_current_char Block

To elaborate more on the form selection process, let's take an example for how this is

done for letter "BAA".



Choosing the correct form of BAA

The previous flowchart is a translation to what is stated in (Appendix A) for BAA letter. Note that we needed to know the surrounding letters up to two steps forward which means two letters after the current one. This was due to that the second form of letter BAA for example to appear, needs that the next character to be HEH and to have a letter after that i.e. the HEH is not the end of the word.

5.1.2 Example for input and output files

Example for an Arabic ASCII file:

```
وقد كان خالد رضي الله عنه يفكر في الإسلام فلما قرأ رسالة
أخيه سرَّ بها سروراً كبيراً وأعجبه مقالة النبي صلى الله عليه
وسلم فيه فتشجع وأسلم

ورأى خالد في منامه كأنه في بلادٍ ضيّقة جديبة فخرج إلى بلد أخضر
واسع فقال في نفسه إن هذه لرؤيا فلمّا قدم المدينة ذكرها لأبي
بكر الصديق فقال له هو مخرجُكَ الذي هداك الله للإسلام والضيقُ الذي
كنتَ فيه من الشرك

الرحلة

يقول خالد عن رحلته من مكة إلى المدينة وددت لو أجد من أصاحب
فلقيت عثمان بن طلحة فذكرت له الذي أريد فأسرع الإجابة وخرجنا
جميعا فأدلجنا سحرا فلما كنا

بالسهل إذا عمرو بن العاص فقال مرحباً بالقوم قلنا وبك قال أين
مسيركم فأخبرناه وأخبرنا أيضاً أنه يريد النبي ليسلم فاصطحبناه
حتى قدمنا المدينة أول يوم من صفر سنة ثمان
```

This file is sent to our C program as an argument when calling it and the output intermediate file is shown at Appendix B.2.

Code shown at Appendix B.3 represents an example for a TEX input file that uses the program output as an input to call the correct letter forms and produce the correct pdf of the text using AlQalam Font.

Here is the output file that is produced by TEX in this case:

وقد كان خالد رضى الله عنه يفكر فى الإسلام فلما قرأ رسالة أخيه سر بها سرورا كبيرا وأعجبه مقاله النبى صلى الله عليه وسلم فيه فتشجع وأسلم
ورأى خالد فى منامه كأنه فى بلاد ضيقة جديبة فخرج إلى بلد أخضر واسع فقال فى نفسه إن هذه لرؤيا فلما قدم المدينة ذكرها لأبى بكر الصديق
فقال له هو مخرج لك الذى هداك الله للإسلام والضيق الذى كنت فيه من الشرك

الرحلة

يقول خالد عن رحلته من مكة إلى المدينة وددت لو أجد من أصاحب فلقيت عثمان بن طلحة فذكرت له الذى أريد فسرع الإجابة وخرجنا
جميعا فدلجنا سحرا فلما كنا
بالهدة إذا عمرو بن العاص فقال مرحبا بالقوم قلنا وبك قال أين مسيركم فخبرناه وأخبرنا أيضا أنه يريد النبى ليسلم فاصطحبناه حتى قدمنا المدينة
أول يوم من صفر سنة ثمان

5.2 Context Analysis Engine Using LuaTEX

LuaTEX is a TEX -based computer typesetting system which started as a version of pdfTEX with a Lua scripting engine embedded. After some experiments it was adopted by the pdfTEX team as a successor to pdfTEX (itself an extension of eTEX , which generates PDFs). Later in the project some functionality of Aleph was included (esp. multi-directional typesetting). The project was originally sponsored by the Oriental TeX project, founded by Idris Samawi Hamid, Hans Hagen, and Taco Hoekwater. The main objective of the project is to provide a version of TEX where all internals are accessible from Lua. In the process of opening up TEX much of the internal code is rewritten. Instead of hard coding new features in TeX itself, users (or macro package writers) can write their own extensions. LuaTEX offers native support for OpenType fonts. In contrast to XeTEX , the fonts are not accessed through the operating system libraries, but through a library based on FontForge. A related project is MPLib (an extended MetaPost library module), which brings a graphics engine into TEX . The LuaTEX team consists of Taco Hoekwater, Hartmut Henkel and Hans Hagen. [21]

LuaTEX Main Features [22]:

- Use of Lua language inside TEX to implement functions.
- Usage of MPLib.
- Use of the callbacks to override default TeX behavior for special cases like our case.

5.2.1 Lua Code Execution

The primitive `\directlua` is used to execute Lua code immediately. The syntax is `\directlua <general text>` The last <general text> is expanded fully, and then fed into the Lua interpreter. We made use of this functionality by porting our context analysis

C function into Lua and load it in order to make use of it instead of using the third party application step. The approach will be shown later in the callback section.

5.2.2 MPLib

By loading the MPLib package using `\usepackage{luamplib}` we can execute:

```
\begin{mplibcode}
beginfig(1);
```

And hence load all the Metafont files of the font (without the 256 letter limitation) and call the Metafont function of any letter with the desired argument values. i.e. change elongation, form,..etc. [23], [24] and [25]

5.2.3 Callbacks Library

This library has functions that register, find and list callbacks.

Callbacks are entry points to LuaTEX 's internal operations, which can be interspersed with additional Lua code, and even replaced altogether. In the first case, TEX is simply augmented with new operations (for instance, a manipulation of the nodes resulting from the paragraph builder); in the second case, its hard-coded behavior (for instance, the paragraph builder itself) is ignored and processing relies on user code only.

More precisely, the code to be inserted at a given callback is a function (an anonymous function or the name of a function variable); it will receive the arguments associated with the callback, if any, and must frequently return some other arguments for TEX to resume its operations. The first task is registering a callback: id, error = callback.register (<string> callback_name, <function> func) id, error = callback.register (<string> callback_name, nil) id, error = callback.register (<string> callback_name, false) where the callback_name is a predefined callback name, see below. The function returns the internal id of the callback or nil, if the callback could not be registered. In the latter case, error contains an error message, otherwise it is nil.

LuaTEX internalizes the callback function in such a way that it does not matter if you redefine a function accidentally.

Callback assignments are always global. You can use the special value nil instead of a function for clearing the callback.

For some minor speed gain, you can assign the boolean false to the non-file related callbacks, doing so will prevent LuaTEX from executing whatever it would execute by default (when no callback function is registered at all). Be warned: this may cause all sorts of grief unless you know exactly what you are doing! This functionality is present since version 0.38 of LuaTEX . We mainly made use of "process_input_buffer" callback. This callback allowed us to change the contents of the line input buffer just before LuaTEX actually starts looking at it.

So, now user can input Arabic text in the TEX file and we can process it to call the correct Metafont function that corresponds to the letter in the context.

5.2.4 Porting the Generic Context Analysis C Code to Lua to use it inside LuaTEX

We have ported the context analysis c code into Lua to load the context analysis function inside LuaTEXand manipulate the input stream using the process_input_buffer callback

from LuaTEX to modify the input that is passed to the TEX engine with the corresponding Metafont call to each input Arabic letter and using our context analysis engine to select the suitable forms of the letters from our font.

These are some notes (or significant difference) that we came a cross when porting the C code to Lua:

In Lua version 5.1:

- Semicolon as a statement separator is optional.
- It supports assignment to a list of variables: local a, b, c = 1, 2, x, y = y, x, or a, b = foo().
- It Supports multiline strings (using [[...]]; can be enclosed with [[...[=[...]=]...]]) and comments (--[[...]]).
- Strings are indexed from 1 rather than 0.
- No integers as a separate numeric type; the number type represent real numbers.
- Non-equality operator is = (for example, if a = 1 then ... end).
- not, or, and keywords used for logical operators.
- Assignments are statements, which means there is no a=b=1 or if (a=1) then ... end.
- No a+=1, a++, or similar shorthand forms.
- No switch statement.
- A control variable in a loop is localized by default and is not available after the loop. Also, changing the value of the control variable inside the loop yields an unexpected behavior.
- Conditionals and other control structures require no brackets.
- Strings and numbers are automatically converted (if string is used where a number is expected and vice versa), but not in equality/inequality.
- return statement can't be used if it's not the last statement in a block.

Here is a code snippet from the original C code and its corresponding port in the Lua code as an example:

C code:

```
string context(string current_line)
{
    string result_line;
    for (size_t i = 0; i < current_line.length(); i=i+2)
    {
        if(i != 0)
        {
                        //Do something.
```

```
        }
    }
    return result_line;
}
```

Corresponding Lua Code:

```
function context(current_line)
result_line= ""
    --In Lua string indices start at index value 1.
        for i = 1, string.len(current_line), 2
        do
                if(i ~= 1) then
                        -- Do something.
                end
        end
    return result_line
end
```

After investigating the callbacks library in LuaTEX we found that the best callback that can help us insert our context analysis engine is the data processing callback that is called "process_input_buffer".

This callback allows us to change the contents of the line input buffer just before LuaTEX actually starts looking at it.

Through using this callback we were able to manipulate the input buffer contents (which contains initially unicode Arabic letters) and generate instead of it the correct MetaPost letter calls from AlQalam base file(originally Metafont calls but we they are still compatible with MetaPost that is introduced in LuaTEX) according to the context.

So the output becomes a series of MetaPost calls to the letter forms.

Encapsulating this output inside `begin{mplibcode}` and `end{mplibcode}` from LuaTEX yields the desired output which is a box that has all the requested letter forms drawn inside and are contextually correct.

5.3 Results

After loading the Lua context analysis code and input the Arabic text using the LuaTEX code shown in Appendix B.4, We were able to get sentences written in AlQalam font easily and without further steps.

The output that corresponds to the input file is shown in Fig. 5.1

Other outputs can be obtained by editing the Arabic text section in the file (inside `\startQalamBuffer` and `\begin{mplibcode}` part)

The next figures starting Fig. 5.2 to Fig. 5.10 show other sample outputs using AlQalam font inside LuaTEX with the ported context analysis algorithm loaded using the command `\directlua{dofile("context.lua")}`

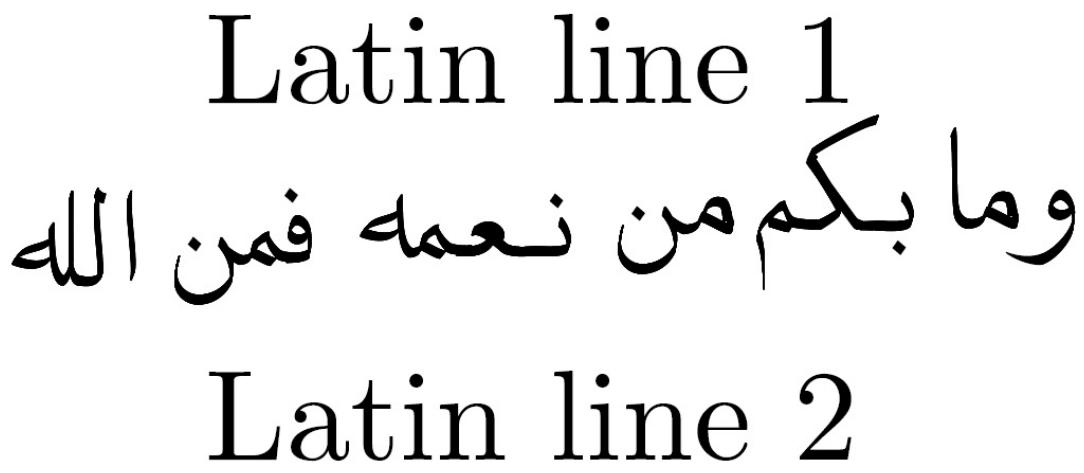


Figure 5.1: Sample output 1

Figure 5.2: Sample output 2

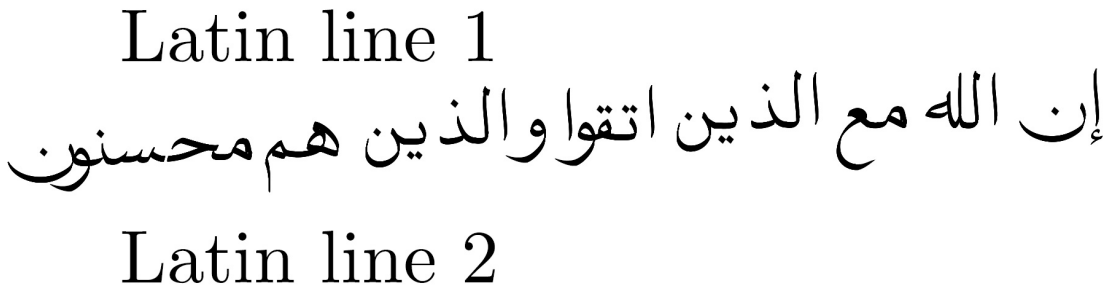


Figure 5.3: Sample output 3

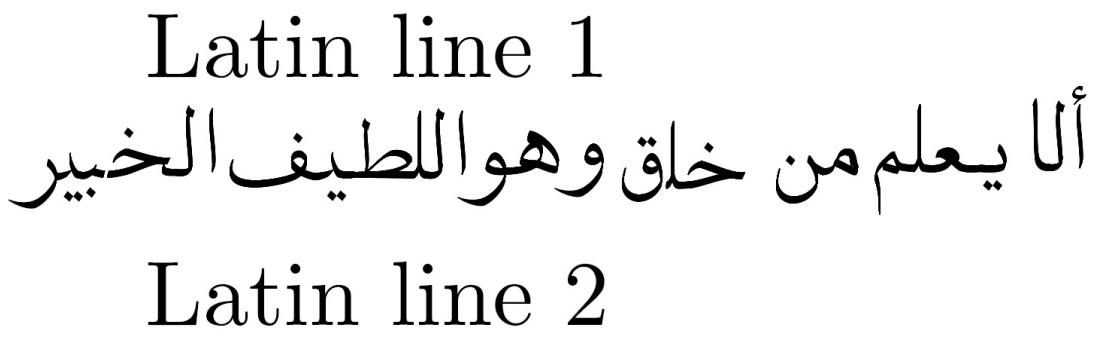


Figure 5.4: Sample output 4

Figure 5.5: Sample output 5

Figure 5.6: Sample output 6

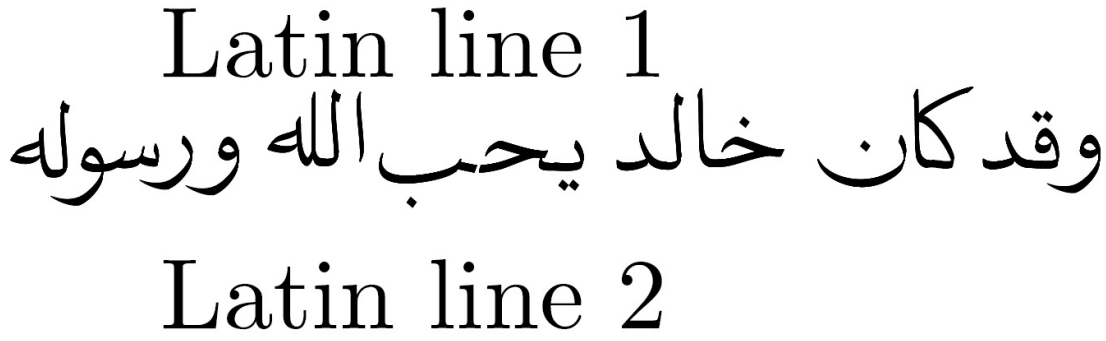


Figure 5.7: Sample output 7

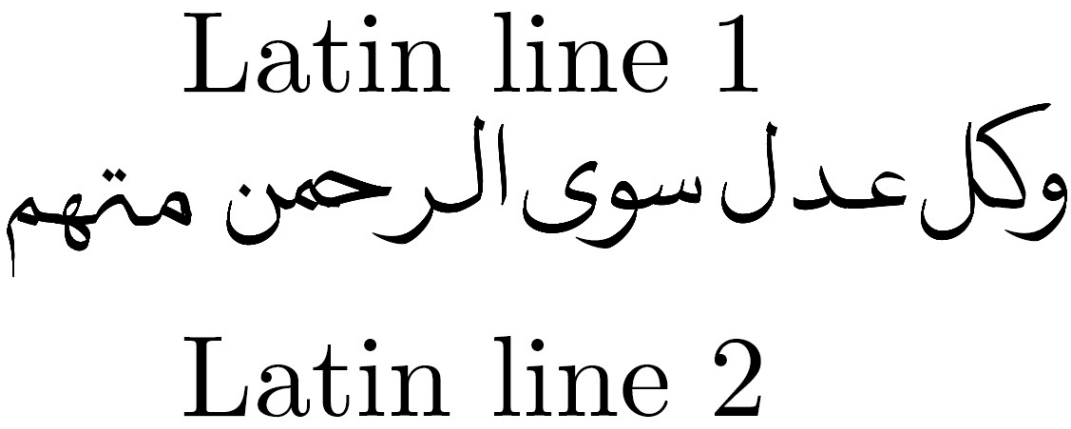


Figure 5.8: Sample output 8

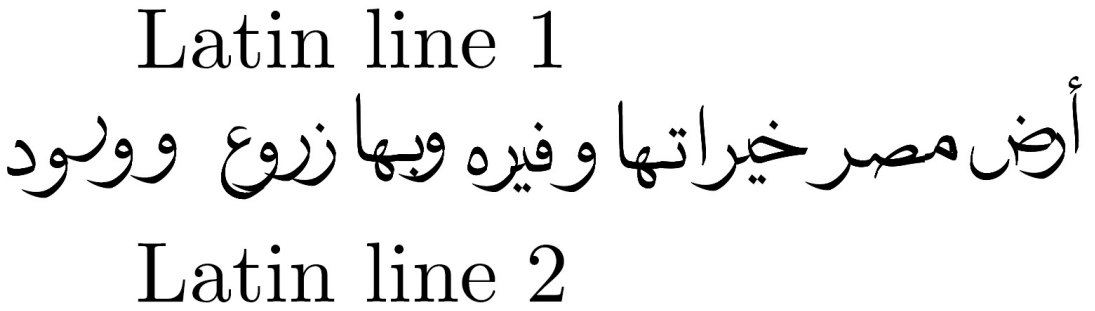


Figure 5.9: Sample output 9

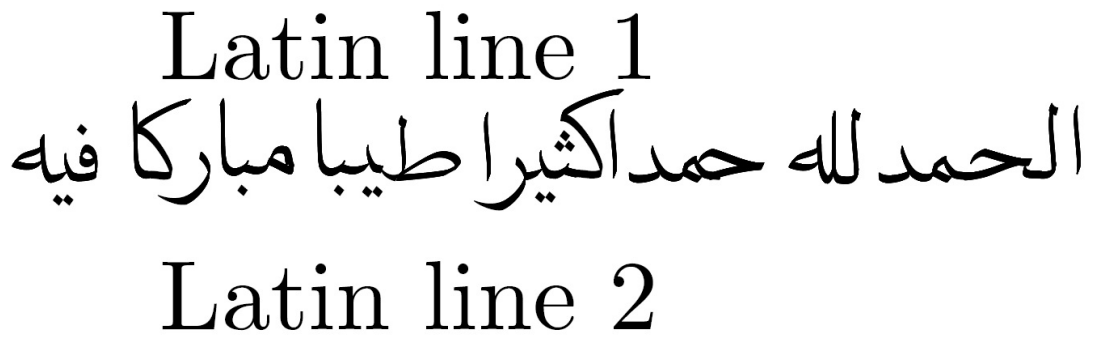


Figure 5.10: Sample output 10

5.4 Evaluation

In this section, we explain the testing methodology used, how the test was designed, and finally, test results and comments.

5.4.1 A Subjective Test

A subjective test was carried out by surveying a sample of 30 people (18 males and 12 females) with ages ranging from 18 to 50 years. The object of the test was to know if the reader will be comfortable with the way out sentences written with AlQalam font using the developed contextual engine looks as compared to sentences written with other Naskh fonts.

5.4.2 Testing Methodology

The test methodology used is that of the Mean Opinion Score (MOS).

This is a subjective test which is very often used to evaluate the perceived quality of media (like audio or video) after compression or transmission in the field of electrical communications and digital signal processing. The MOS is expressed as a single number in the range of 1 to 5, where 1 is lowest quality and 5 is highest quality.

5.4.3 Design of the Test

In our case, the MOS is a numerical indication of the perceived quality of a written sentence. A survey was designed asking a reader to evaluate sentences in terms of their written quality and he is asked to rate them in terms of how comfortable he is with them. A rating of 1 is given to low quality and uncomfortable sentences, and a rating of 5 is given to a sentence of high written quality and which is comfortable to the reader.

The test was composed of eight sentences, each written in four different Naskh fonts:

- Simplified Arabic
- Arabic Typesetting
- Amiri Font
- AlQalam font using the generic contextual analysis engine presented in this thesis

The eight sentences were chosen to test these main features:

- Each letter is correctly generated without compatibility problems with MetaPost
- Choosing the correct forms of letters (context analysis engine test)
- Connections between letters (joining problems that require modification in Metafont files)
- Kerning

For a more reliable and unbiased test, the order of fonts used was varied in consecutive rows and all sentences are set to approximately the same sizes although same point sizes of different fonts were not exactly equal. Finally, the MOS is calculated as the arithmetic mean of all the individual scores. The generated sentences used in the designed test are shown below.

إن مع العسر يسرا

وقد كان خالد يحب الله و رسوله

إن الله مع الذين اتقوا و الذين هم محسنون

الشعر لسان العرب

فسيكفيكهم الله

و كل عدل سوى الرحمن متهم

الحمدلله حمدا كثيرا طيبا مباركا فيه

أرض مصر خيراتها وفيرة و بها زروع و ورود

Figure 5.11: MOS sentences generated using Simplified Arabic

إن مع العسر يسرا

وقد كان خالد يحب الله و رسوله

إن الله مع الذين اتقوا و الذين هم محسنون

الشعر لسان العرب

فسيكفيكهم الله

و كل عدل سوى الرحمن متهم

الحمدلله حمدا كثيرا طيبا مباركا فيه

أرض مصر خيراتها وفيرة و بها زروع و ورود

Figure 5.12: MOS sentences generated using Arabic Typesetting

إن مع العسر يسرا

وقد كان خالد يحب الله و رسوله

إن الله مع الذين اتقوا و الذين هم محسنون

الشعر لسان العرب

فسيكفيكهم الله

و كل عدل سوى الرحمن متهم

الحمدلله حمدا كثيرا طيبا مباركا فيه

أرض مصر خيراتها وفيرة و بها زروع و ورود

Figure 5.13: MOS sentences generated using Amiri Font

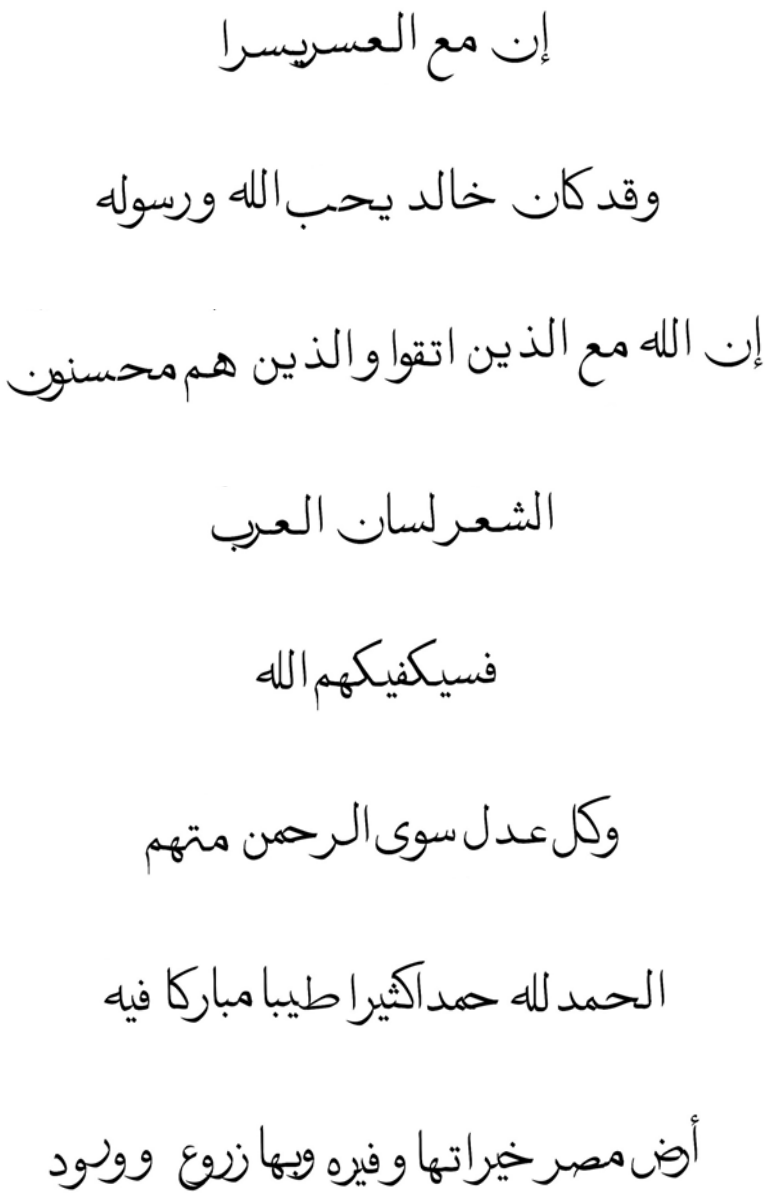


Figure 5.14: MOS sentences generated using AlQalam Font

5.4.4 Test Results

Sentence No.	Simplified Arabic	Arabic Typesetting	Amiri Font	AlQalam Font
1	3	3.6	3.8	2.9
2	2.6	3.9	4.1	4
3	2.1	3.7	4.1	3.9
4	2.5	3.9	4.2	2.7
5	2.2	3.7	4.2	3.3
6	3	4.1	4.5	2.9
7	3.1	3.4	3.9	2.8
8	2.3	3.9	3.5	2.1
MOS	2.3	3.8	4	3.1

5.4.5 Comments and Conclusion on Results

The results show that the font in a fair condition as some sentences were poorly perceived and others were well perceived. This is due to some issues that appears in the poorly perceived sentences which are:

- In sentence one: The space between the third and fourth words is too tight to be a space between two words. This was due to the kerning parameters of "RAA" letter.
- Sentences two and three were well perceived as they have almost no issues except for a small diffraction from the third sentence against the line.
- The same kerning issue of "RAA" appears in the fourth sentence but here within the same word and this caused the sentence to be poorly perceived.
- Some issues appeared in the sixth and seventh sentences like wrong forms of "TEH" and "HAA" were combined so the last word in the sixth sentence appeared in inappropriate way. Also, in the seventh sentence, the space between "ALEF" in the third word and "KAF" in the fourth word should be longer to clarify the starting of a new word. In addition to a noticeable issue in joining "TAA" with "YAA" in the fifth word.
- The last sentence had the lowest score as it has:

  1. "DAD" letter in the first word should be fixed as it has joining issues.
  2. Kerning between "WAW" and "BAA" in the fifth word.
  3. The kerning in the sixth word should be revisited as "WAW" was expected to be at higher level on te line than its previous letter "RAA".
  4. Same applies for the kerning issues in the last word, levels of "WAW" and "RAA" should be fixed.

We believe that more work is still ahead enhancing the font and this became now much easier due to the latest enhancements in the font usage scheme. This will eventually enable fast discovery of bugs and also fast testing and development cycles.

Chapter 6

Conclusion and future work

6.1 Conclusion

In this thesis, we introduced a new algorithm for the contextual analysis of Arabic characters that are part of rich Arabic fonts. We used AlQalam font as our test font, ported our generic context analysis C code to Lua language, loaded it inside LuaTEX , overrode the normal input buffer TEX operations to manipulate the buffer contents and do our custom context analysis work to get sentences written in AlQalam font with the correct letter forms according to the context using a normal LuaTEX editor to input Arabic text.

With few changes, this algorithm can be used for any rich Arabic font to handle its context analysis process.

This enabled us to recognize more font issues easier than ever, so this engine can be used to test rich fonts like AlQalam font and point out if there are missing forms, issues with specific forms or joining issues quickly and easily. Also, it will be released with the final version of the font to override the default context analysis engine of TEX and act as the custom context analysis engine that is tailored to satisfy the rich Arabic font needs and hence show its beauty and richness.

This is a step towards facilitating the creation of rich Arabic fonts that avoid the disadvantages of the most fonts and methods listed in chapter 2 and offer:

- Ease of use.
- Direct Arabic text input.
- Ability to use various beautiful forms of letters.
- One package that includes Arabic letters and mathematical symbols.

6.2 Future Work

As we have reached a state that we do context analysis for the font and generate the correct Arabic words with fixed elongations and without the usage of ligatures, we now need to override the default line-breaking algorithm of TEX (that uses spaces only to justify lines) to use another algorithm that makes use of the various ways of justifying lines for Arabic rich fonts. See Figure: 6.1

Currently the appended Metapost calls that returns from the context analysis Lua code are sent to Metapost to construct the line. The constructed line returns as a Metapost box

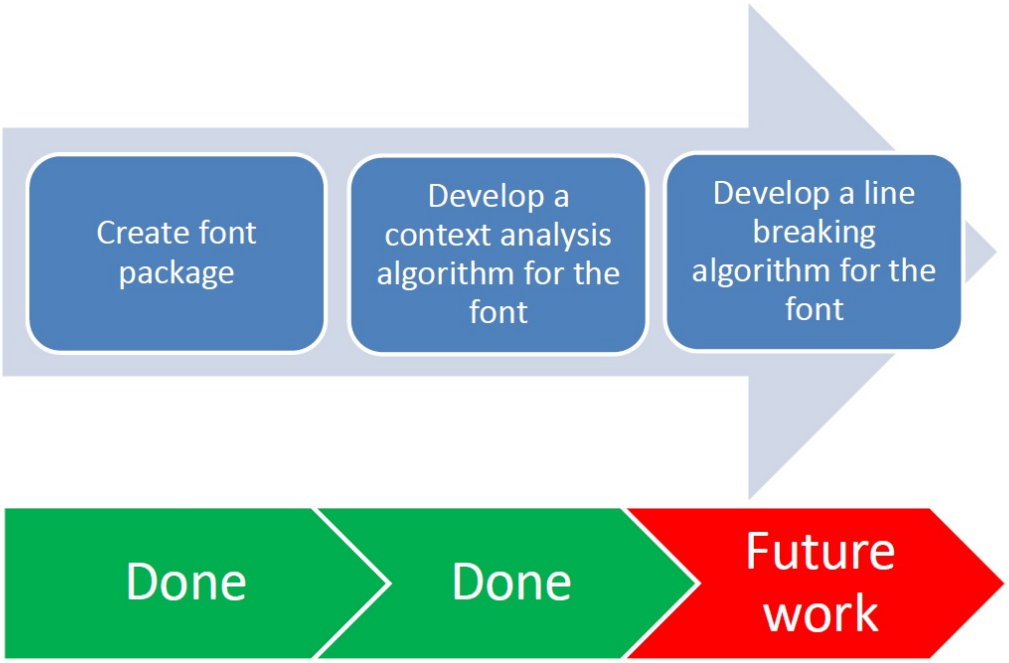


Figure 6.1: Work Done and Future Work

to the TEX engine. There should be a higher level algorithm that constructs a single line using Metapost calls and send it to Metapost engine in LuaTEX and then get the box width of the Arabic line and checks if the width is within the acceptable range or not. In case of the box is out of the acceptable range, another iteration should be done (by adding or removing words from the line according to the desired increase/decrease to box width). This is a very primitive line breaking algorithm.

A more advanced line breaking algorithm will take into consideration the elongation features of the font, available ligatures and combinations, changing the letter forms and using the default behavior for Latin which is stretching by changing space widths while taking the breaking decisions. [26]

Hyphenation also can be used for Uighur language which is written using Arabic script and uses hyphenation. [27]

This would be performed through the usage of LuaTEX 's callbacks feature to override the main algorithm. (See Chapter 4 from [21])

Example for the LuaTEX callbacks that can be used:

Node list processing callbacks like:

- buildpage_filter

  This callback is called whenever LuaTEX is ready to move stuff to the main vertical list. This callback can be used to do specialized manipulation of the page building stage like imposition or column balancing.

- pre_linebreak_filter

  This callback is called just before LuaTEX starts converting a list of nodes into a stack of `\hboxes`, after the addition of `\parfillskip`.

- linebreak_filter

This callback replaces LuaTEX 's line breaking algorithm.

- post_linebreak_filter

  This callback is called just after LuaTEX has converted a list of nodes into a stack of `\hboxes`.

- hpack_filter

  This callback is called when TEX is ready to start boxing some horizontal mode material. Math items and line boxes are ignored at the moment.

- vpack_filter

  This callback is called when TEX is ready to start boxing some vertical mode material. Math displays are ignored at the moment. This function is very similar to the hpack_filter.

- pre_output_filter

  This callback is called when TEX is ready to start boxing the box 255 for `\output`.

- hyphenate

  This callback has to insert discretionary nodes in the node list it receives. Setting this callback to false will prevent the internal discretionary insertion pass.

- ligaturing

  This callback has to apply ligaturing to the node list it receives.

- kerning

  This callback has to apply kerning between the nodes in the node list it receives.

- mlist_to_hlist

  This callback replaces LuaTEX 's math list to node list conversion algorithm.

A complete set of callbacks can be used consecutively to construct paragraphs inside LuaTEX overriding the default behavior of TEX and enabling the usage of a full custom approach to fulfill rich fonts needs. [28]

References

[1] A. Gulzar and S. ur Rahman, ``Nastaleeq: A challenge accepted by omega,'' *TUGboat*, vol. 29, no. 1, pp. 89--94, 2007.

[2] A. S. Zayed, *Ahdath Al-turuq Leta`leem Al-khotot Al-`arabiya [New methods for learning Arabic calligraphy]*. Cairo, Egypt: Maktabat ibn-Sina, 1990.

[3] *The Holy Qur'an*. Madinah, KSA: King Fahd Complex for Printing the Holy Qur'an, 1986.

[4] H. A. H. Fahmy, ``AlQalam for typesetting traditional Arabic texts,'' *TUGboat*, vol. 27, pp. 159--166, Jan. 2007.

[5] K. Lagally, ``ArabTeX Typesetting Arabic with vowels and ligatures,'' in *EuroTEX '92: Proceedings of the 7th European TEX Conference, Prague, Czechoslovakia, September 14--18, 1992* (J. Zlatuška, ed.), Proceedings of the European TEX Conference, (Brno, Czechoslovakia), pp. 153--172, Masarykova Universita, Sept. 1992.

[6] A. M. Sherif and H. A. H. Fahmy, ``Meta-designing parameterized Arabic fonts for AlQalam,'' *TUGboat*, vol. 29, pp. 435--443, Jan. 2008.

[7] A. Lazrek, ``RyDArab Typesetting Arabic mathematical expressions,'' *TUGboat*, vol. 25, no. 2, pp. 141--149, 2004.

[8] M. Hafiz, ``Mathfont.'' `http://mhafiz.deyaa.org/mathfont.html`. Accessed: 2015-03-01.

[9] A. Azmi and A. Alsaiari, ``Arabic typography: A survey,'' *International Journals of Engineering and Sciences IJENS*, vol. 9, no. 10, pp. 16--22.

[10] A. Lazrek, ``Vers un système de traitement du document scientifique arabe,'' Master's thesis, Faculté des Sciences Semlalia, Université Cadi Ayyad, Marrakech, Morocco, 2 2002.

[11] J. Kew, ``The xetex project: typesetting for the rest of the world,'' *XIV Ogólnopolska Konferencja Polskiej Grupy Uzytkowników Systemu TeX*, 2006.

[12] J. Braams, ``Babel, a multilingual package for use with latex's standard document classes.'' `http://www.phys.ethz.ch/~ihn/latex/babel.pdf`. Accessed: 2015-03-01.

[13] Y. Jabri, ``The Arabi system - TeX writes in Arabic and Farsi ,'' *TUGboat*, vol. 27, no. 2, pp. 147--153, 2006.

[14] K. Hosny, ``Amiri font.'' http://www.amirifont.org/. Accessed: 2015-03-01.

[15] A. E. Sergeous and H. Gawish, *Al-Mo'aser in Mathematics*. Cairo, Egypt: Maktabat Al-Talaba For Printing, Publishing and Distribution, 1999.

[16] R. Ierusalimschy, W. Celes, and L. H. de Figueiredo, ``Lua Official Website.'' http://www.lua.org/about.html.

[17] K. F. Moshfeghi, ``A new algorithm for contextual analysis of farsi characters and its implementation in java,'' *17th International Unicode Conference*, 9 2012.

[18] D. E. KNUTH and M. F. PLASS, ``Breaking paragraphs into lines,'' *SOFTWARE-PRACTICE AND EXPERIENCE*, vol. 11, pp. 1119--1184, Feb. 1981.

[19] A. Sherif, ``Parameterized arabic font development for computer typesetting systems,'' Master's thesis, FACULTY OF ENGINEERING, CAIRO UNIVERSITY, GIZA, EGYPT, 1 2008.

[20] U. Consortium, ``The Unicode Standard 7.0 for Arabic.'' http://unicode.org/charts/PDF/U0600.pdf. Accessed: 2015-03-01.

[21] T. Hoekwater, ``Luatex reference manual v0.79.'' http://www.luatex.org/svn/trunk/manual/luatexref-t.pdf. Accessed: 2015-03-01.

[22] T. Hoekwater, ``Luatex,'' *TUGboat*, vol. 28, no. 3, pp. 312--313, 2007.

[23] T. Hoekwater and H. Hagen, ``Mplib: Metapost as a reusable component,'' *TUGboat*, vol. 28, no. 3, pp. 317--318, 2007.

[24] T. Hoekwater, ``Metapost developments: Mplib project report,'' *TUGboat*, vol. 29, no. 3, pp. 380--382, 2008.

[25] H. Hagen, ``The metapost library and luatex,'' *TUGboat*, vol. 29, no. 3, pp. 446--453, 2008.

[26] M. J. E. Benatia, M. Elyaakoubi, and A. Lazrek, ``Arabic text justification,'' *TUGboat*, vol. 27, pp. 137--146, Jan. 2006.

[27] Y. Haralambous, ``Infrastructure for high-quality arabic typesetting,'' *TUGboat*, vol. 27, pp. 167--175, Jan. 2006.

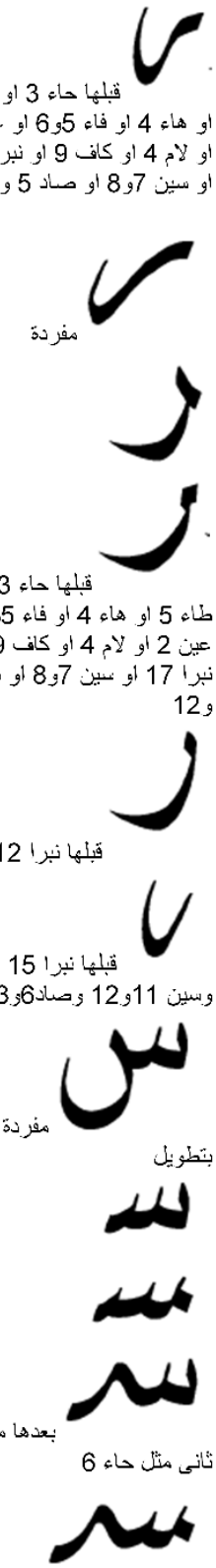
[28] P. Isambert, ``Luatex : What it takes to make a paragraph,'' *TUGboat*, vol. 32, no. 1, pp. 68--76, 2011.

Appendix A

Context Analysis Map for AlQalam Font

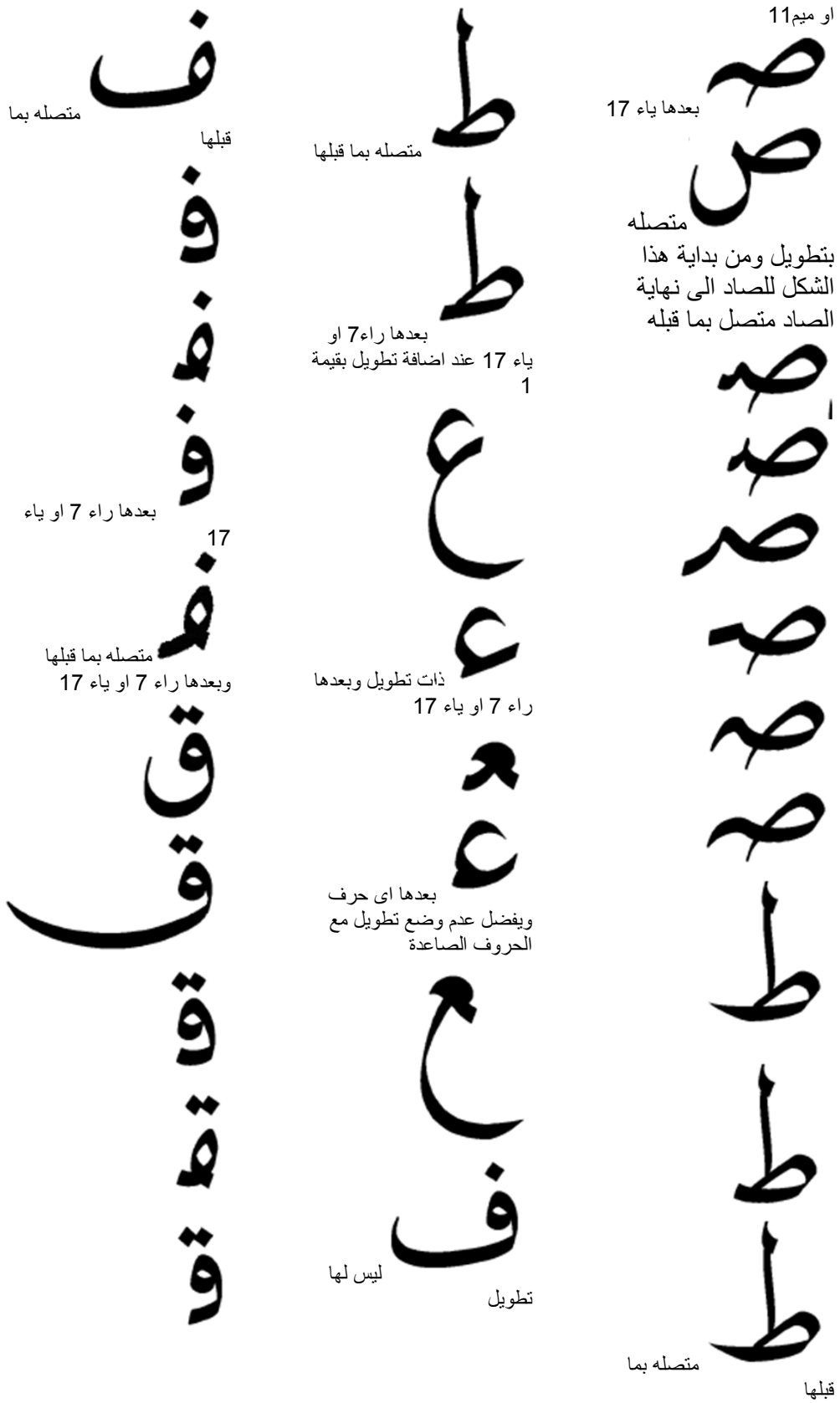
راء 2

مستوى ثانى
أو ثالث
مستوى
ثانى وبعدها مستوى ثالث
بعدها حروف
صاعدة مثل الف 2 ولام 7
بعدها ميم9



قبلها حاء 3 او طاء 5
او هاء 4 او فاء 5و6 او عين 2
او لام 4 او كاف 9 او نبرا 17
او سين 7و8 او صاد 5 و12
مفردة
قبلها حاء 3 او
طاء 5 او هاء 4 او فاء 5و6 او
عين 2 او لام 4 او كاف 9 او
نبرا 17 او سين 7و8 او صاد 5
و12
قبلها نبرا 12
قبلها نبرا 15 و18
وسين 11و12 وصاد6و13
مفردة
بتطويل
بعدها
مستوى ثانى مثل حاء 6

بعدها راء 5
بعدها ياء 17
بعدها راء7 او
ميم 11
متصله
بتطويل
مفردة
بتطويل
بعدها
مستوى ثانى مثل حاء 6
بعدها راء 5
بعدها راء7


او ميم11
بعدها ياء 17
متصله
بتطويل ومن بداية هذا الشكل للصاد الى نهاية الصاد متصل بما قبله
متصله بما قبلها
بعدها راء7 او ياء 17 عند اضافة تطويل بقيمة 1
ذات تطويل وبعدها راء 7 او ياء 17
بعدها اى حرف ويفضل عدم وضع تطويل مع الحروف الصاعدة
ليس لها تطويل
متصله بما قبلها
متصله بما قبلها
بعدها راء 7 او ياء 17
متصله بما قبلها وبعدها راء 7 او ياء 17

متصله بما قبلها
متصله بما قبلها
تحتاج الى اضافة تطويل
بدون تطويل
وبعدها الحروف الصاعدة
تحتاج الى اضافة تطويل
بعدها راء 2
ذات تطويل
وبعدها اى حرف
مع ميم9 او ياء
17 او راء2
بعدها هاء 5 او ميم 10
مثل 3 لكن بدون حليه
تحتاج الى اضافة تطويل
لام وحاء
متصلين
بعدها حروف صاعدة
بعدها ياء 17او ميم11
او ميم 6

قبلها لام 10
متصلة بما قبلها
قبلها عين2 او كاف
9
قبلها لام5
قبلها نبرا 15 و18
او سين 11و12 او صاد 6 و
13
متصلة بما قبلها
وعدها ياء 17
مبتدأة وبعدها ياء
17

مفردة بتطويل
متصله بتطويل
متصلة وبتطويل
وقبلها نبرا 7
بعدها راء 2او
ياء 17
قبلها لام 5
قبلها نبرا 15 و18
وبعدها ياء 17

بتطويل وقبها نبرا 15
و 18
متصلة
منفردة بتطويل
متصله بتطويل
وقبلها اى حرف نازل

لأ إلا الله لله ء أ ؤ

ؤ ء ئ ى ئ د ئ

Appendix B

Code Segments

B.1 TEX Code to Generate Boxes Around Letters

```
\begin{lstlisting}[frame=single]

\magnification=1000

\global\font\mytest=qalam

%See exercise 11.5 of the TeXbook

%% Horizontal rule of height #1 and depth #2
\def\hidehrule#1#2{\kern-#1%
 \hrule height#1 depth#2 \kern-#2 }

%% Vertical rule of width (#1+#2)
%% with #1 to the left and #2 to
%% the right of the reference point
\def\hidevrule#1#2{\kern-#1{\dimen0=#1
 \advance\dimen0 by#2\vrule width\dimen0}\kern-#2 }

%% Print the edges of a box using rules of given thickness
%% the first argument is the thickness outside the box
%% the second argument is the thickness inside the box
\def\makeblankbox#1#2{\hbox{\lower\dp0\vbox{\hidehrule{#1}{#2}% top hrule
 \kern-#1 % overlap the rules at the corners
 \hbox to \wd0{\hidevrule{#1}{#2}%
   \raise\ht0\vbox to #1{}% set the vrule height
   \lower\dp0\vtop to #1{}% set the vrule depth
   \hfil\hidevrule{#2}{#1}}% closes the hbox with the two vrules
 \kern-#1\hidehrule{#2}{#1}}}}% bottom hrule
%%closes the vbox then hbox then the def

%% A definition to write a character within its bounding box
%% Sets box0 to the character
%% then makes a horizontal box with the following contents:
%% a blank box whose dimensions are that of box0
%% a kern with negative width of box0
%% (i.e. move left with that width in order to get back to the start)
%% a `baseline'
%% a kern with negative width of box0
%% then print box0 (the character) on top of the blankbox

%  Set the contents of the box
\def\boxchar#1{\setbox0=\hbox{#1}
% leave vertical mode (see exercise 13.1 of TeXbook)
  \leavevmode
  \hbox to \wd0{% start an hbox with width equal to that of the box
        \makeblankbox{0.1pt}{0pt}% make the blank box
        \kern-\wd0%  move back by the width of the box
        \vrule width \wd0 height 0.05pt depth 0.05pt% `baseline'
        \kern-\wd0%  move back by the width of the box
        \box0}% put the character and close the box
  } % close the def


%% For Latin characters
\def\l#1{\boxchar{#1}}

%% Temporary workaround the font of AlQalam
\def\q#1{\boxchar{{\mytest\char#1}}}
```

```
\pagedir TRT
\bodydir TRT

\pardir  TLT
\textdir TLT
%\pagedir TRT\pardir TRT\bodydir TRT\textdir TRT

\l{a}\l{b}\l{p}\l{q}\l{w}\l{g}\l{!}\l{1}\l{2}

abpqwg!12

\q{0}\q{190}\q{99}\q{62}\q{201}\q{223}
\q{0}\q{190}\q{99}\q{62}\q{238}\q{202}

\q{0}\l{a}\q{9}\l{a}\q{9}\l{a}

\q{1}

\pardir  TRT
\textdir TRT

\l{a}\l{b}\l{p}\l{q}\l{w}\l{g}\l{!}\l{1}\l{2}

abpqwg!12


\q{0}\q{190}\q{99}\q{62}\q{201}\q{223}
\q{0}\q{190}\q{99}\q{62}\q{238}\q{202}

\q{0}\l{a}\q{9}\l{a}\q{9}\l{a}

\q{1}

\q{0} \q{1} \q{2} \q{3} \q{4} \q{5} \q{6}
\q{7} \q{8} \q{9} \q{10} \q{11} \q{12} \q{13}
\q{14} \q{15} \q{16} \q{17} \q{18} \q{19} \q{20}
\q{21} \q{22} \q{23} \q{24} \q{25}

\q{26} \q{27} \q{28} \q{29} \q{30} \q{31} \q{32}
\q{33} \q{34} \q{35} \q{36} \q{37} \q{38} \q{39}
\q{40} \q{41} \q{42} \q{43} \q{44} \q{45} \q{46}
\q{47} \q{48} \q{49} \q{50}

\q{51} \q{52} \q{53} \q{54} \q{55} \q{56} \q{57}
\q{58} \q{59} \q{60} \q{61} \q{62} \q{63} \q{64}
\q{65} \q{66} \q{67} \q{68} \q{69} \q{70} \q{71}
\q{72} \q{73} \q{74} \q{75}

\q{76} \q{77} \q{78} \q{79} \q{80} \q{81} \q{82}
\q{83} \q{84} \q{85} \q{86} \q{87} \q{88} \q{89}
\q{90} \q{91} \q{92} \q{93} \q{94} \q{95} \q{96}
\q{97} \q{98} \q{99} \q{100}

\q{101} \q{102} \q{103} \q{104} \q{105} \q{106}
\q{107} \q{108} \q{109} \q{110} \q{111} \q{112}
\q{113} \q{114} \q{115} \q{116} \q{117} \q{118}
\q{119} \q{120} \q{121} \q{122} \q{123} \q{124}
```

```
\q{125}

\q{126} \q{127} \q{128} \q{129} \q{130} \q{131}
\q{132} \q{133} \q{134} \q{135} \q{136} \q{137}
\q{138} \q{139} \q{140} \q{141} \q{142} \q{143}
\q{144} \q{145} \q{146} \q{147} \q{148} \q{149}
\q{150}

\q{151} \q{152} \q{153} \q{154} \q{155} \q{156}
\q{157} \q{158} \q{159} \q{160} \q{161} \q{162}
\q{163} \q{164} \q{165} \q{166} \q{167} \q{168}
\q{169} \q{170} \q{171} \q{172} \q{173} \q{174}
\q{175}

\q{176} \q{177} \q{178} \q{179} \q{180} \q{181}
\q{182} \q{183} \q{184} \q{185} \q{186} \q{187}
\q{188} \q{189} \q{190} \q{191} \q{192} \q{193}
\q{194} \q{195} \q{196} \q{197} \q{198} \q{199}
\q{200}

\q{201} \q{202} \q{203} \q{204} \q{205} \q{206}
\q{207} \q{208} \q{209} \q{210} \q{211} \q{212}
\q{213} \q{214} \q{215} \q{216} \q{217} \q{218}
\q{219} \q{220} \q{221} \q{222} \q{223} \q{224}
\q{225} \q{226} \q{227} \q{228} \q{229} \q{230}
\q{231} \q{232} \q{233} \q{234} \q{235} \q{236}
\q{237} \q{238} \q{239} \q{240} \q{241} \q{242}
\q{243} \q{244} \q{245} \q{246} \q{247} \q{248}
\q{249} \q{250} \q{251} \q{252} \q{253} \q{254}

\end
```

B.2 Example Output from the Context Analysis C Application

```
%%Line Done!


%%WAW
\q{234}%
%%QAF
\q{175}%
%%DAL
\q{89}%
%%space
\ %%new word
%%KAF
\q{182}%
%%ALEF
\q{1}%
%%NOON
\q{220}%
%%space
\ %%new word
%%KHAH
\q{76}%
%%ALEF
\q{1}%
%%LAM
\q{188}%
%%DAL
\q{89}%
%%space
\ %%new word
%%REH
\q{94}%
%%DAD
\q{131}%
%%YEH
\q{249}%
%%space
\ %%new word
%%ALEF
\q{0}%
%%LAM
\q{188}%
%%LAM
\q{187}%
%%HEH
\q{226}%
%%space
\ %%new word
%%AIN
\q{156}%
%%NOON
\q{210}%
%%HEH
\q{226}%
%%space
\ %%new word
%%YEH
\q{237}%
%%FEH
```

and these Metafont calls goes on till the end of file.

```
%%TEH_MARBUTA
\q{226}%
%%space
\ %%new word
%%ALEF_WITH_HAMZA_ABOVE
\q{2}%
%%WAW
\q{234}%
%%LAM
\q{186}%
%%space
\ %%new word
%%YEH
\q{237}%
%%WAW
\q{235}%
%%MEEM
\q{199}%
%%space
\ %%new word
%%MEEM
\q{200}%
%%NOON
\q{221}%
%%space
\ %%new word
%%SAD
\q{122}%
%%FEH
\q{169}%
%%REH
\q{97}%
%%space
\ %%new word
%%SEEN
\q{111}%
%%NOON
\q{210}%
%%TEH_MARBUTA
\q{226}%
%%space
\ %%new word
%%THEH
\q{35}%
%%MEEM
\q{201}%
%%ALEF
\q{1}%
%%NOON
\q{220}%
```

B.3 TEX Code that Consumes the Context Analysis C Application Output

```
%\include{varinclude.tex}
\magnification=1000

\global\font\mytest=qalam
%\global\font\mytest=qalam scaled 4340
%\global\font\mytest=qalam scaled 1085
%\global\font\mytest=qalam scaled 3000
%\global\font\mytest=qalam scaled 1440

%See exercise 11.5 of the TeXbook

%% Horizontal rule of height #1 and depth #2
\def\hidehrule#1#2{\kern-#1%
 \hrule height#1 depth#2 \kern-#2 }

%% Vertical rule of width (#1+#2)
%% with #1 to the left and #2 to the right of the reference point
\def\hidevrule#1#2{\kern-#1{\dimen0=#1
 \advance\dimen0 by#2\vrule width\dimen0}\kern-#2 }

%% Print the edges of a box using rules of given thickness
%% the first argument is the thickness outside the box
%% the second argument is the thickness inside the box
\def\makeblankbox#1#2{\hbox{\lower\dp0\vbox{\hidehrule{#1}{#2}% top hrule
 \kern-#1 % overlap the rules at the corners
 \hbox to \wd0{\hidevrule{#1}{#2}%
   \raise\ht0\vbox to #1{}% set the vrule height
   \lower\dp0\vtop to #1{}% set the vrule depth
   \hfil\hidevrule{#2}{#1}}% closes the hbox with the two vrules
 \kern-#1\hidehrule{#2}{#1}}}}% bottom hrule
                              %%closes the vbox then hbox then the def

%% A definition to write a character within its bounding box
%% Sets box0 to the character
%% then makes a horizontal box with the following contents:
%% a blank box whose dimensions are that of box0
%% a kern with negative width of box0
%% (i.e. move left with that width in order to get back to the start)
%% a `baseline'
%% a kern with negative width of box0
%% then print box0 (the character) on top of the blankbox

\def\boxchar#1{\setbox0=\hbox{#1}%  Set the contents of the box
  \leavevmode% leaves vertical mode (see exercise 13.1 of TeXbook)
  \hbox to \wd0{% start an hbox with width equal to that of the box
        \makeblankbox{0pt}{0pt}% make the blank box
        \kern-\wd0%  move back by the width of the box
        \vrule width \wd0 height 0.05pt depth 0.05pt% `baseline'
        \kern-\wd0%  move back by the width of the box
        \box0}% put the character and close the box
  } % close the def


%% For Latin characters
%\def\l#1{\boxchar{#1}}

%% Temporary workaround the font of AlQalam
\def\q#1{\boxchar{{\mytest\char#1}}}
```

```
%\def\q#1{\mytest\char#1}
%\def\q#1{\rule{0pt}{1ex}{\mytest\char#1}}
%\def\q#1{\hfill{\mytest\char#1}}%\par\bigskip\bigskip\bigskip\bigskip}
%\def\q#1{{\mytest\char#1}}%\par\bigskip\bigskip\bigskip\bigskip}

\pagedir TRT
\bodydir TRT

\pardir  TLT
\textdir TLT


\pardir  TRT
\textdir TRT



\input{output}



\end
```

B.4 LuaTEX Code that Loads the Context Analysis Lua Code and Outputs Text Using AlQalam Font

```
\documentclass[a4paper]{article}
\usepackage{luamplib}

\begin{document}

%%%context analysis test using lua code%%%

%%% Here we use the directlua command to excute the dofile command, which loads the context
analysis lua code into memory.
%%% and then define a new lua function that is called doContextAnalysis() that accepts an
input line (arabic utf-8 letters) and returns
%%% metafont calls to be used inside mplib section to draw the correponding and correct
forms of the letters according to their position in the context.
\directlua{
dofile("context.lua")
function doContextAnalysis(line)
    line = context(line)
    return line % this returns the line to LuaTeX's input buffer
end
}

\directlua{
function doProcessCallParameters(str) %for future use to switch modes (context analysis only
or with line breaking)
    % do something useful with the parameter to \startQalamBuffer{.....}
    end
}

%%%startQalamBuffer is a macro to switch on processing the special processing of input
(context analysis and calling the correct metafont forms)
%%% this macro calls at first doProcessCallParameters() function with is not used for now
and can be used in the future to process inputs before passing them to
%%% the context analysis and line breaking engines
%%% and then it calls callback.register() function which overrides the default process\_
input\_ buffer operations of TeX with our doContextAnalysis() function operations
\def\startQalamBuffer#1{
\directlua{
     doProcessCallParameters(#1)
     callback.register("process_input_buffer",doContextAnalysis)
}
}

\def\stopQalamBuffer{
\directlua{
     callback.register("process_input_buffer",nil)}
} %dummy macro to switch off processing the input by cancelling the overriding of the input
buffer processing.
%%%%%%%end context analysis test%%%%%%


%%%%%%%%%%Usage Example%%%%%%%%%%%%

%%%Call the startQalamBuffer macro to initlize the environment. The section between start
and endQalamBuffer will be processed using our lua functions
\startQalamBuffer{figure1}
Latin line 1

%%This section is required to start the mplib code execution.
%% The arabic text will be substituted with metafont (metapost compatible) fucntoin calls
```

```
and will be executed by the embedded metapost engine inside LuaTeX.
\begin{mplibcode}
beginfig(1);

%% the next three includes are essntial to define Qalam font specifications and available
character set.
input header_mp_luatex
input qalam_mp_luatex
input qalambase

%% Enter Arabic Text here:

و ما بكم من نعمة فمن الله

endfig;
\end{mplibcode}

Latin line 2

\end{document}
```

Appendix C

Code Documentation

C.1 File Sets

C.1.1 Font files

AlQalam font was developed using Metafont and it is supplied as a set of files which are:

1. Header files: header.mf and qalam.mf. These two files contain the definitions of the font pen and global variables and function definitions that will be used in drawing most of the letters.

2. Character files: alef.mf, baa.mf,…These files can be compiled using the script shown in section 4.1 to produce one file which will contain all the character definitions and will be called qalambase.mf

C.1.2 Context Analysis C code

The developed generic context analysis algorithm was written in C/C++ before porting it to Lua to be used with AlQalam.

The source code was consisting of:

1. context.cpp: This file contain the main() function that performs the context analysis. It accepts an plain Arabic text file and produce another plain text file that contains calls to the correct forms of letters from the tfm font package. It includes two files: codes.h and forms.h

2. codes.h: This file contains #defines (or constant variable definitions) of Arabic letters and their unicode representations.
   Example: #define ALEF (signed int)0xfffffa7.
   These definitions are separated in this file for easy maintenance of the code and to avoid using hardcoded numbers inside the core code which increases the readability and keeps the code modular as possible.

3. forms.h: This file contains mapping between letter forms and their index in the created font package (tfm file). Example lines: #define alefone 0
   #define aleftwo 1
   #define alefthree 2
   #define aleffour 3

#define aleffive 4
#define alefsix 5
#define alefseven 6

This file is also created to avoid hardcoding, increase readability, reusability and modularity.

The output file resulting from running this module should be included in a tex file that uses the tfm font package in order to produce a pdf document written using the target font.

The context analysis algorithm itself is described in section 5.1.1.

C.1.3 LuaTEX files

The context analysis C code was ported to Lua in order to be used with AlQalam font and LuaTEX .
The file set contain:

1. context.lua: This file contains the core function that performs the context analysis which is:
   function context(current_ line)

   This function accepts a buffer of plain text characters (Arabic and Latin) and returns a string of Metafont calls to the correct Arabic letter forms that corresponds to the inputs according to their position in the context. (It doesn't make any processing on the Latin letters and outputs them as is).
   The file also contains a mapping section between the letter form names and their corresponding Metafont function calls.
   Examples:
   alefone = "alef(1,1)"
   aleftwo = "alef(2,1)"
   alefthree = "alef(3,1)"
   aleffour = "alef(4,1)"
   aleffive = "alef(5,1)"
   alefsix = "alef(6,1)"
   alefseven = "alef(7,1)"

   and contains the mapping section to define mapping between Arabic letters and ligatures and their unicode representations to increase readability.

   The context analysis algorithm itself is described in section 5.1.1.

2. mp_ base.tex: This file is the centralized file that loads the context.lua file using the `\directlua` command:

```
\directlua{
dofile("context.lua")
function doContextAnalysis(line)
    line = context(line)
    return line % this returns the line
                % to LuaTeX's input buffer
end
}
```

and overrides the input buffer operations of TEX using the callbacks registrations.

```
\directlua{
      doProcessCallParameters(#1)
      callback.register("process_input_buffer"
                        ,doContextAnalysis)
}
```

It also has a MetaPost section that receives the processed text using the context analysis code and includes all the required font files to produce text using AlQalam font.

```
\begin{mplibcode}
beginfig(1);

%% the next three includes are
%% essential to define Qalam font
%% specifications and available character set.
input header_mp_luatex
input qalam_mp_luatex
input qalambase

%% Enter Arabic Text here:

endfig;
\end{mplibcode}
```

6.2 Supporting Additions to the Font Character Set

In this section we will discuss how to add new characters, ligatures and symbols (like math symbol sets) to the LuaTEX developed files so we can use them as part of the font.

Actually, this became a lot easier with the LuaTEX approach which provided us with the ability to include Metafont files inside mplipcode code segments.

So, there will be no need to make any further steps nor create font packages and so on. All what is required is include the new character definition in the mplib section using:
input newchar.mf
and then issuing function calls to the character with the correct arguments inside the section.
Example:

```
\begin{mplibcode}
beginfig(1);

input header_mp_luatex
input qalam_mp_luatex
input qalambase
input newchar

%% Call the new character function here:

endfig;
\end{mplibcode}
```

To no to increase the input list, the script shown in section 4.1 may be used in a directory full of Metafont character definition files to compile them into a single file and then user have to include this single file into LuaTEX in the mplib section inputs.

6.3 Fixing Font Issues

Several issues were found during producing documents using AlQalam font:

1. Missing characters in the output: By tracing the context analysis code and finding which letter and which form should be used (this also can be performed by looking at the font tables as the one in Appendix A.), we can go the character definition section in the Metafont files then tracing the error in the form number (if exists) using Metafont interpreter. If the form doesn't exist in the character set, it should be described using Metafont and added to the character set. More information about the Metafont interpreter can be found in the Metafont Manuals and tutorials.

2. Kering issues: If the spacing between a letter like "RAA" or "WAW"...etc and their consecutive letters is not compliant with the font specifications and standards, a change in the "end point" of the character is required.
   The "end point" defines where the consecutive character should start to be drawn relative to the current one.

3. Some parts of the letter is wider than the rest of the parts: This may be due to scaling of the pen used to draw this part. We faced and issue with the "shazya" part

of "RAA" and "WAW" where it appeared wider than the body of the letter. So we fixed this by adjusting the scale of the pin drawing the "shazya".

4. Wrong forms of letters: This may be due to the form number in Metafont character definition is different to what is stated in the table in Appendix A.
   Or may be a bug in the context analysis code and the condition that meets the input context leads to calling a wrong form of the letter.

ملخص البحث

يعد هذا البحث استكمالا للأعمال السابقة في مشروع القلم من جهة تطوير و اختبار الخط. و حتي نتمكن من الحصول علي حزمة للخط قابلة للاستخدام من قبل برمجيات الكتابة، بدأنا ببناء حزمة لخط "القلم" باستخدام ميتافونت. بناء الحزمة تضمن تصحيح و تصليح مشاكل وجدت في ملفات ميتافونت الخاصة ببعض الحروف لجعلها قابلة للتشغيل.

بعد أن حصلنا علي حزمة جاهزة للخط، انتقلنا بها لتجربتها تحت بيئة تخ. في المراحل الأولي من التجربة كنا نقوم بنداء الحروف داخل تخ بطريقة يدوية أي عن طريق بصمة كل حرف في حزمة الخط و هو ما ساعدنا علي اكتشاف المزيد من المشاكل في الخط و التي تحتاج إلي تصحيح و تحسين مثل مشاكل في التوصيل بين الحروف و مشاكل تتعلق بعدم وجود بعض أشكال الحروف. و بناء عليه بدأنا مرحلة جديدة من تصحيح الأخطاء و إضافة الأشكال المفقودة.

بعد ذلك قمنا بتطوير برنامج مستقل وسيط بلغة السي لتحليل السياق أوتوماتيكيا و مناداة بصمات الحروف الصحيحة المناسبة لأماكنها في النص بطريقة آلية يناسب حزم الخطوط العربية الثرية بشكل عام.

بعد تجربة البرنامج مع خط القلم أردنا دمج البرنامج بطريقة ما مع الخط لتسهيل استخدام الخط بدون استخدام برامج وسيطة و المرور بالكثير من الخطوات للحصول علي ملفات مكتوبة بخط القلم.

و تبعا لذلك قمنا باستكشاف المميزات المتاحة في بيئة لواتخ ووجدنا توافر الإمكانات المطلوبة بها لتلبية متطلباتنا حيث أنها تمكننا من استخدام لغة لوا لكتابة برامج مساعدة للخط و استخدامها في تجاوز السلوك الافتراضي لبيئة تخ في التعامل مع الخطوط.

و من هنا قمنا بترقية برنامج تحليل السياق من لغة سي إلي لغة لوا و تشغيله داخل لواتخ و تجاوزنا السلوك الإفتراضي لبيئة تخ في التعامل مع الحروف المدخلة لإجراء تحليل السياق المناسب لخط القلم و بذلك تمكنّا من الحصول علي جُمَل مكتوبة بخط القلم مُراعي فيها استخدام الأشكال الصحيحة للحروف تبعا للسياق باستخدام محرر نصوص لواتخ لإدخال النص العربي و بدون خطوات إضافية.

مهنـــــــــدس: شريف سمير حسن منصور

تاريـخ الميـــلاد: 1986/02/09

الجنسيـــــــة: مصري

تاريخ التسجيل: 2010/10/01

تـــاريخ المنـــح: / /

القســـــــــم: هندسة الإلكترونيات و الإتصالات الكهربية

الدرجـــــــــة: ماجستير العلوم

المشرفون : أ.م.د. حسام علي حسن فهمي

الممتحنـــــون : أ.م.د. حسام علي حسن فهمي (المشرف)

أ.د. محسن عبدالرازق رشوان (الممتحن الداخلي)

أ.د. السيد مصطفي سعد (الممتحن الخارجي)

عنـــوان الرسالـة :

تكوين و تنضيد الخطوط العربية بطريقة آلية

الكلمات الدالة : -

اللغة العربية، خطوط، تخ، القلم، تحليل السياق.

ملخـــص البحـــث :

يعد هذا البحث استكمالا للأعمال السابقة في مشروع القلم من جهة تطوير و اختبار الخط.

حتي نتمكن من الحصول علي حزمة للخط قابلة للاستخدام من قبل برمجيات الكتابة، بدأنا ببناء حزمة لخط "القلم" باستخدام ميتافونت و قمنا بحل المشاكل التي قابلناها حينما استخدمناها لكتابة وثائق باللغة العربية.

بعد ذلك قمنا بتطوير برنامج مستقل وسيط بلغة السي لتحليل السياق أوتوماتيكيا و مناداة بصمات الحروف الصحيحة المناسبة لأماكنها في النص بطريقة آلية يناسب حزم الخطوط العربية الثرية بشكل عام و قمنا بترقيته إلي لغة لوا لاستخدامه مع خط "القلم".

و تم دمج برنامج تحليل السياق المكتوب بلغة لوا مع الخط باستخدام الميزات الجديدة المتاحة في بيئة لواتخ لتمكين المستخدمين من الحصول علي نص عربي باستخدام خط "القلم" بسرعة و سهولة.

تكوين و تنضيد الخطوط العربية بطريقة آلية

إعداد

شريف سمير حسن منصور

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة

كجزء من متطلبات الحصول على درجة ماجستير العلوم

في

هندسة الإلكترونيات و الإتصالات الكهربية

يعتمد من لجنة الممتحنين:

الأستاذ المساعد: حسام علي حسن فهمي — المشرف

الأستاذ الدكتور: محسن عبد الرازق رشوان — الممتحن الداخلي

الأستاذ الدكتور: السيد مصطفي سعد — الممتحن الخارجي
(أستاذ دكتور بهندسة حلوان)

كلية الهندسة – جامعة القاهرة

الجيزة – جمهورية مصر العربية

٢٠١٥

تكوين و تنضيد الخطوط العربية بطريقة آلية

إعداد

شريف سمير حسن منصور

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة

كجزء من متطلبات الحصول على درجة ماجستير العلوم

في

هندسة الإلكترونيات و الإتصالات الكهربية

تحت إشراف

أ.م.د. حسام علي حسن فهمي

قسم هندسة الإلكترونيات و الإتصالات الكهربية

كلية الهندسة – جامعة القاهرة

كلية الهندسة – جامعة القاهرة

الجيزة – جمهورية مصر العربية

٢٠١٥

تكوين و تنضيد الخطوط العربية بطريقة آلية

إعداد

شريف سمير حسن منصور

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة

كجزء من متطلبات الحصول على درجة ماجستير العلوم

في

هندسة الإلكترونيات و الإتصالات الكهربية

كلية الهندسة – جامعة القاهرة

الجيزة – جمهورية مصر العربية

٢٠١٥